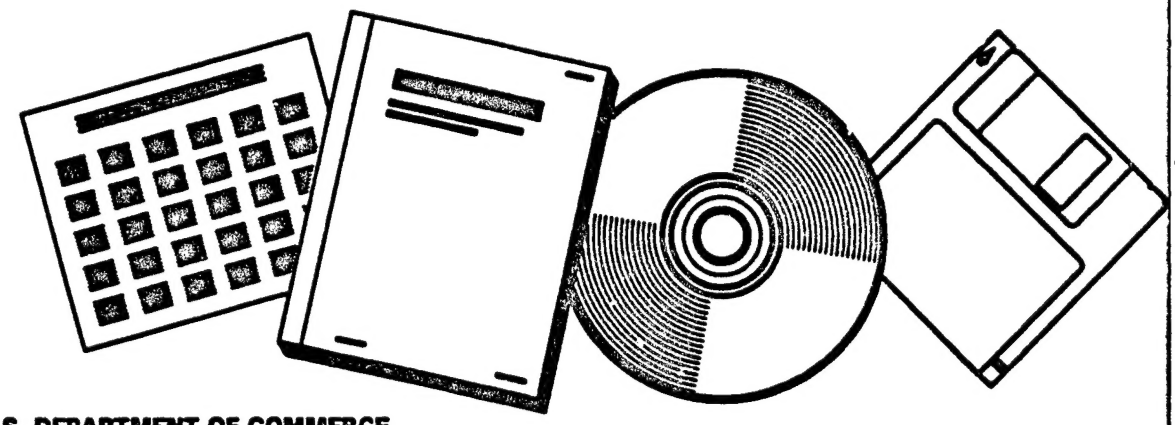
PB96-148788

LABELLED FORMAL LANGUAGES AND THEIR USES

STANFORD UNIV., CA

MAY 91



U.S. DEPARTMENT OF COMMERCE
National Technical Information Service

BIBLIOGRAPHIC INFORMATION

PB96-148788

Report Nos: STAN-CS-83-982

Title: Labelled Formal Languages and Their Uses.

Date: cMay 91

Authors: D. H. Greene.

Performing Organization: Stanford Univ., CA. Dept. of Computer Science.

Sponsoring Organization: *National Science Foundation, Washington, DC.

Contract Nos: NSF-MSC-83-00984, NSF-IST-820-1926

Type of Report and Period Covered: Doctoral thesis.

NTIS Field/Group Codes: 62B (Computer Software)

Price: PC A08/MF A02

Availability: Available from the National Technical Information Service, Springfield, VA. 22161

Number of Pages: 155p

Keywords: *Programming languages, *Formalism, *Grammars, *Permutations, Algorithms, Trees(Mathematics), Computations, String theory, Theses.

Abstract: This research augments formal languages with the machinery necessary to describe labelled combinatorial objects such as trees, permutations, and networks. The most attractive feature of this method of describing combinatorial objects is the direct translation to generating functions. Treating the grammar of an ordinary formal language as a set of equations and then solving these equations yields an enumerating generating function. This is still true of labelled formal languages although the equations are usually differential rather than rational or algebraic. There are two promising applications for labelled formal languages. In the analysis of algorithms one often identifies combinatorial quantities that can be described with labelled formal languages and, using the translation mentioned above, these quantities can be easily computed. The other application uses labelled formal languages to control a general-purpose system for the ranking, sequencing, and selection of combinatorial objects. Both of these applications demonstrate the value of labelled formal languages as a descriptive and analytic tool.

June 1983

Report No. STAN-CS-83-982

Labelled Formal Languages and Their Uses

by

Daniel H. Greene

Department of Computer Science

Stanford University
Stanford, CA 94305

Labelled Formal Languages and Their Uses

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

by
Daniel Hill Greene
June 1983

This research and/or preparation was supported in part by the National Science Foundation under grants MSC-83-00984, IST-820-1926, and by the Systems Development Foundation.

ABSTRACT

This research augments formal languages with the machinery necessary to describe labelled combinatorial objects such as trees, permutations, and networks. These objects typically have requirements for their labels (trees, for example, can be equivalent under permutation of subtrees) that make certain labellings invalid or redundant. To deal with this problem, formal languages are augmented with partial orders—derived strings have partial orders specifying acceptable labellings, and productions of the grammar contain fragments of partial orders. The traditional rewrite step in a derivation is now coupled with a substitution that joins two partial orders.

The most attractive feature of this method of describing combinatorial objects is the direct translation to generating functions. Treating the grammar of an ordinary formal language as a set of equations and then solving these equations yields an enumerating generating function. This is still true of labelled formal languages although the equations are usually differential rather than rational or algebraic.

There are two promising applications for labelled formal languages. In the analysis of algorithms one often identifies combinatorial quantities that can be described with labelled formal languages and, using the translation mentioned above, these quantities can be easily computed. The other application uses labelled formal languages to control a general-purpose system for the ranking, sequencing, and selection of combinatorial objects. Both of these applications demonstrate the value of labelled formal languages as a descriptive and analytic tool.

ACKNOWLEDGMENTS

Many people contributed to the preparation and debugging of this work. I would like to thank Phyllis Winkler and Frank Yellin, who were both generous with their help typesetting and formatting, and to numerous readers, Jeff Ullman, Lyle Ramshaw, Andrei Broder, Yoram Moses, and Harry Mairson, whose comments clarified the material of this dissertation.

It is probably easiest to say that my adviser, Don Knuth, did not make the paper used to print this dissertation, because his assistance can be seen in every other aspect of the work. He designed the fonts used throughout the text with the METAFONT system, and his TEX system enabled me to typeset mathematical formulas with relative ease. The Pascal code in the appendices is written in WEB, another system of Don's, that allows for the decomposition of programs into modules, and the natural intermingling of exposition with code. But I am most grateful to Don for his personal example; as an excited researcher, pioneer, and kind guider of graduate students, he has done a lot to make my years at Stanford challenging and enjoyable.

Undergirding all of this was the longstanding support of my parents, who encouraged me in a way that was remarkably free of pressure, and in a way that helped me find work that was truly satisfying. It is to them that I dedicate this dissertation.

To my parents
Edmund Greene and Janet Schuyler Hubley Greene

TABLE OF CONTENTS

FIGURES

CHAPTER 1

INTRODUCTION

The remarkably simple concept of a grammar has led to considerable clarity in the development of computer science.

Definition. *A grammar is a four-tuple (T, N, S, P) consisting of*

1) a terminal alphabet T (usually small letters);

2) a nonterminal alphabet N (usually capital letters);

3) a start symbol $S \in N$;

4) a set of productions P.

The heart of a grammar is item 4, the set of productions that guide the generation of strings.

Definition. *A production is a rewrite rule with two strings over the alphabet $T \cup N$. The left hand string is separated from the right hand string by an arrow, for example, $AB \rightarrow cDeF$.*

Definition. *A derivation step consists of matching a substring with the left hand string of a production and then rewriting it with the right hand string.*

Definition. *A string is derived by a grammar if it is possible to obtain the string from the start symbol of the grammar with a finite number of derivation steps.*

In the remainder of this work, grammars will always be context-free, that is, their productions will always have a single nonterminal on the left hand side. We will usually call these "grammars" even though they are more properly called "context-free grammars" in the study of formal languages. It is often convenient to group several productions with a common left hand side using vertical bars to separate the production possibilities. So, for example, $A \rightarrow B \mid C \mid D$ will denote three productions $A \rightarrow B$, $A \rightarrow C$, and $A \rightarrow D$. The greek ϵ is used as a symbol for the null string.

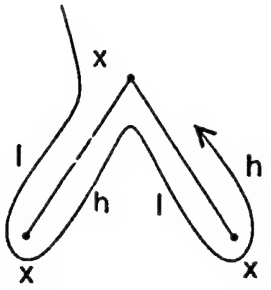


Figure 1.1 The Encoding of Trees

Grammars are used primarily for description; they specify a set of acceptable strings, called a formal language. For example, if we are encoding trees by their walks, with l for lower, h for higher, and x for a tree node as shown in Figure 1.1, then the grammar

$$\begin{aligned} T &\rightarrow xF \\ F &\rightarrow lThF \mid \epsilon \end{aligned} \tag{1.1}$$

specifies the language of valid trees. The first production means that every tree is a root node connected to a forest of offspring, while the second production defines a forest as a sequence of trees. Starting with the symbol T, the tree of Figure 1.1 is derived as follows:

$$\begin{aligned} &T \\ &xF \\ &xlThF \\ &xlxFhF \\ &xlxhF \\ &xlxhlThF \\ &xlxhlxFhF \\ &xlxhlxhF \\ &xlxhlxh\,. \end{aligned} \tag{1.2}$$

Considerable research has been devoted to the inversion of this process (parsing a string to obtain a derivation) and to the study of the descriptive power of formal languages.

The focus of this dissertation is on another interesting property of formal languages. If the productions of a grammar are treated as equations, with $\rightarrow$ replaced by $=$, $\mid$ by $+$, and ϵ by 1, and if the equations are solved for the start symbol, the result is a generating function for the number of derivations of the grammar.

Generating functions have become a central tool of combinatorial mathematics; they are used to study sequences of numbers $\{g_i\}_{i\geq 0}$. The g_i are implanted as the Taylor coefficients of a function G of a dummy variable x:

$$G(x) = \sum g_i x^i\,, \tag{1.3}$$

and then the properties of the sequence are studied through the properties of $G(x)$. Generating functions have various flavors. We will be using ordinary generating functions like (1.3) in this chapter, and exponential generating functions in the following chapters. An exponential generating function has the form

$$G(x) = \sum g_i \frac{x^i}{i!}. \tag{1.4}$$

In our applications, the g_i coefficients will count the number of derivations of a grammar.

Grammar (1.1) converts to two equations:

$$\begin{aligned} T &= xF \\ F &= lThF + 1. \end{aligned} \tag{1.5}$$

We solve for T by algebraically eliminating all other nonterminal characters and by treating terminal characters as dummy variables.

$$\begin{aligned} F &= \frac{1}{1 - lhT} \\ T - lhT^2 &= x \\ T &= \frac{1 \pm \sqrt{1 - 4lhx}}{2lh}. \end{aligned} \tag{1.6}$$

The extraneous root is discarded with the observation that the grammar cannot derive an empty tree, and so $T(0)$ must be zero. The remaining root is expanded:

$$\begin{aligned} T &= \frac{1 - \sqrt{1 - 4lhx}}{2lh} \\ &= \sum_{j \geq 1} \binom{1/2}{j} (-1)^{j-1} 2^{2j-1} l^{j-1} h^{j-1} x^j \\ &= \sum_{j \geq 1} \frac{1}{j} \binom{2(j-1)}{j-1} l^{j-1} h^{j-1} x^j. \end{aligned} \tag{1.7}$$

We obtain a multivariate generating function where the coefficient of $l^a h^b x^c$ is the number of strings having a l's, b h's, and c x's. However, l and h are not particularly useful; there will always be one less of these walk control characters than there are tree nodes. Hereafter we shall drop such irrelevant characters by replacing them with 1 earlier in the analysis. We conclude that the number of trees with j nodes is equal to the Catalan number $\frac{1}{j}\binom{2(j-1)}{j-1}$.

This close connection between formal languages and power series appeared in the early work of Schützenberger and Chomsky [Schützen 1961] [Chomsky 1963]. They found that power series over noncommutative variables could be very helpful in the classification of formal languages: regular expressions became rational equations, and context free languages became algebraic equations. Issues of ambiguity were now questions about the coefficients of power series. A full discussion of this line of research can be found in the recent work of Salomaa and Soittola [Salomaa 1978].

Maurice Gross was responsible for taking this line of research in the direction of enumeration of combinatorial objects [Gross 1966]. He realized that by commuting the variables of the power series, one could collect in the coefficients all words with the same composition of terminal characters. (In the example above, we collected together all trees with the same number of nodes.) This meant that if a combinatorial object could be described with a string of characters, and if the valid strings could be described with a grammar, then it was easy to obtain a generating function for the family of combinatorial objects. The grammar had to be relatively simple (regular, or context free) and had to derive unambiguously the language of valid strings; nevertheless, this technique provided straightforward enumerations for several complex combinatorial families.

The aim of this dissertation is to extend the applicability of formal languages to a wide range of labelled combinatorial objects. This extension is developed in Chapter 2. Chapters 3 and 4 explore the advantages of the extension, using it to analyze algorithms and to generate combinatorial objects. But first of all, the remainder of this chapter is devoted to a survey of the enumerative uses of formal languages, in order to describe techniques that will be helpful in later chapters and to understand the limitations of formal languages.

1.1 Variations of Trees.

In a fundamental sense, every grammar describes a family of trees. (These are the derivation trees for the valid strings of the language.) So it is not surprising that context free languages prove useful in the enumeration of objects having an underlying tree structure, even though on the surface the problem may be unrelated to trees. Maurice Gross gives the following example: A ternary triangulation is formed by repeatedly dividing single triangles into three parts, where the division is accomplished by adding a new vertex in the interior of the triangle and connecting it to the three corners. A sample triangulation is shown in Figure 1.2.

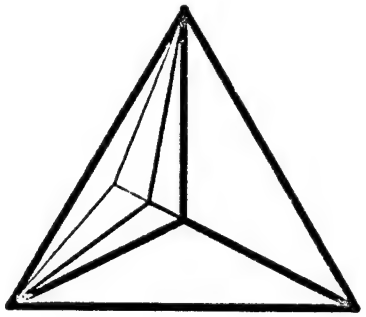


Figure 1.2 A Ternary Triangulation

Triangulations are encoded with parentheses: Each triangle is represented by a balanced pair of parentheses, ( ), and when a triangle is subdivided three new pairs are introduced inside the original, (( ) ( ) ( )), representing the new triangles in clockwise order, starting with the triangle on the base of the enclosing triangle. The base edges of the

new triangles are those edges in common with the sides of the old triangle. Following this procedure, the triangulation pictured in Figure 1.2 is encoded as

$$\Big(\,()\,\big(\,(()()())\,()\,()\,\big)\,()\,\Big).$$

The grammar for this language guarantees that the interior of each pair of parentheses will be divided into three parts or none at all:

$$S \to (SSS) \mid (). \tag{1.8}$$

We would like to count triangulations according to the number of triangles, so "(" is mapped to z, and ")" is mapped to 1:

$$S = zS^3 + z. \tag{1.9}$$

Rather than solve this cubic equation for S, there is an easier technique, due to Lagrange, that allows us to find the coefficients of S without knowing a closed form for S itself. In general, if we have a functional equation for S,

$$S = zf(S), \tag{1.10}$$

and if we let $\langle z^n\rangle S$ denote the coefficient of z^n in the Taylor series for S, then Lagrange's inversion formula gives us:

$$\langle z^n\rangle\, S = \frac{1}{n}\langle S^{n-1}\rangle\, f(S)^n. \tag{1.11}$$

Further discussion of this formula can be found in Appendix A (which is oriented towards exponential generating functions as opposed to the ordinary generating functions of this chapter), and in the next example of this section. For the current example we obtain:

$$\begin{aligned}\langle z^n\rangle\, S &= \frac{1}{n}\,\langle S^{n-1}\rangle (1+S^3)^n \\ &= \begin{cases} \dfrac{1}{n}\binom{n}{\frac{n-1}{3}} & n = 3j+1 \\ 0 & \text{otherwise.}\end{cases}\end{aligned} \tag{1.12}$$

For further enumeration problems with underlying tree structure see the works of Maurice Gross [Gross 1966], Jay Goldman [Goldman 1978 and 1979], W. Kuich [Kuich 1970a and 1970b], and Vaughan Pratt [Knuth 1973; problem 2.2.1–11]. Several other problems, although not analyzed originally with formal languages, also exhibit tree structure: Schröder's second problem, Fibonacci sequences, and occupancy problems (with unlabelled balls placed in an ordered collection of boxes) can all be encoded with context free languages.

The pervasive tree structure suggests that we analyze the most general tree possible, cast in the form of functional composition [Goldman 1979]:

$$\begin{aligned}
T &\to F_0 \mid F_1 \mid F_2 \mid \dots \\
F_0 &\to f_{01} \mid f_{02} \mid \dots \mid f_{0c_0} \\
F_1 &\to f_{11}(T) \mid f_{12}(T) \mid f_{13}(T) \mid \dots \mid f_{1c_1}(T) \\
F_2 &\to f_{21}(T,T) \mid f_{22}(T,T) \mid \dots \mid f_{2c_2}(T,T) \\
F_3 &\to f_{31}(T,T,T) \mid f_{32}(T,T,T) \mid \dots \mid f_{3c_3}(T,T,T) \\
&\vdots
\end{aligned} \tag{1.13}$$

A term, T, is either a constant like f_{01}, or a function applied to several terms such as $f_{21}(T,T)$. The first index of an f symbol indicates the number of operands expected by the function. In order to count all possible compositions according to the number of terms present, we map the f's onto x. This leaves an implicit equation

$$T = xC(T) \tag{1.14}$$

where $C(T)$ has coefficients equal to the number of functions of each degree:

$$C(T) = \sum_{j \geq 0} c_j T^j . \tag{1.15}$$

Lagrange's inversion formula is well adapted to invert an equation like (1.14):

$$\langle x^N \rangle T = \frac{1}{N} \langle T^{-1} \rangle \left(\frac{C(T)}{T} \right)^N . \tag{1.16}$$

But here Raney has developed the following, alternate, approach to the problem that gives an interesting constructive interpretation of Lagrange's formula [Raney 1960]. First, notice that no information is lost by dropping the parentheses from a composition of functions,

$$\begin{aligned}
&f_{10}(f_{21}(f_{01}, f_{02})) \\
&f_{10}\, f_{21}\, f_{01}\, f_{02} ,
\end{aligned} \tag{1.17}$$

because the functions appear in an unambiguous Polish, or prefix order. However, not every list of functions is a valid composition; we cannot always add parentheses and obtain a single term.

When is a list of functions a valid composition? The answer uses the following scheme of Lukasiewicz: Functions requiring j operands are given a weight of $j - 1$, so the total weight of a complete term will always be -1. Continuing the example above, $f_{10} f_{21} f_{01} f_{02}$ has weights of 0, 1, -1, and -1 for a total of -1, as expected.

Suppose that the we jumble together a collection of f's with a total weight of -1, is this necessarily a valid composition? Not always, but curiously one (and only one) of the cyclic permutations of our list will be a valid composition. The reason lies in an additional

constraint on the weights. Not only must they sum to -1, but a cumulative, left to right sum of the weights must remain greater than -1 until the last f. This avoids a premature completion of the term.

If we graph the sum of the weights of the first j terms as a function of j, then a typical composition looks something like Figure 1.3 below.

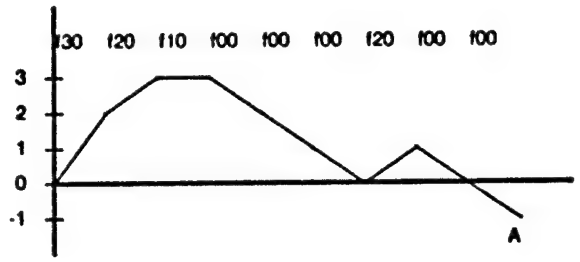


Figure 1.3 The Cumulative Weights of a Functional Composition

Notice that any cyclic permutation of these letters cannot be a valid composition since point A would have a y component less than or equal to -1, and would appear before the last position in the permutation. Conversely, if we take any collection of f's summing to -1, find the point with smallest y component (breaking ties in favor of the leftmost point), and cyclically shift this point to the last position, then the new graph will remain positive until the end, indicating a valid composition.

To derive Lagrange's inversion formula we simply interpret the last few paragraphs with generating functions. Starting with $C(T)$, which has coefficients c_j equal to the number of functions taking j operands, we can weight the functions according to Lukasiewicz by dividing by T: $C(T)/T$.

Next we select N functions with total weight -1,

$$\langle T^{-1} \rangle \left(\frac{C(T)}{T} \right)^N , \tag{1.18}$$

and we know that only one cyclic permutation will be correct, so there is a $1/N$ chance that we have a valid compositional pattern:

$$\langle x^N \rangle T = \frac{1}{N} \langle T^{-1} \rangle \left(\frac{C(T)}{T} \right)^N . \tag{1.19}$$

This is Lagrange's formula for its inversion of $T = xC(T)$.

The functional composition problem explains why Lagrange's formula is so often seen in the solution of grammar related equations. By adjusting c_i, the number of functions with i operands, we are controlling the branching of a tree. Many problems are special cases of the last analysis. For example the triangulation grammar appearing at the beginning

of this section branches three ways ($c_3 = 1$) and there is only one way to terminate the branching, ($c_0 = 1$). The remaining coefficients are zero. This means that our earlier application of Lagrange's inversion formula in equation (1.12) is equivalent to treating functional composition of a ternary function and a constant.

The generality of the last result also points to a limitation in the use of grammars in enumeration. The problems may be complicated by several nonterminals (corresponding to constraints on the composition of functions) but it is hard to get beyond the basic tree-like nature of these grammars. Nevertheless, some extensions are possible. The next few sections are devoted to techniques that increase the versatility of context free languages.

1.2 Indexing and the Q Operator

Consider the following problem:

Problem. *Count the number of partitions of n into sequences of positive integers $i_1 \le i_2 \le i_3 \le \cdots \le i_m$ such that $i_1 + i_2 + \cdots + i_m = n$, and the parts of the partition are bounded both in number ($m \le j$) and in size ($i_m \le k$).*

To solve this problem, partitions are encoded with blocks of q's separated by x's and t's. The q's encode the parts of the partition, the x's mark the end of each block, and the t's signal transitions in block size. Using this system, a partition of q into $1+1+3+4$ would appear as

$$tqxqxttqqqxtqqqqx. \tag{1.20}$$

However, it is impossible to express a string like (1.20) with a finite grammar, since the grammar could not insure that block sizes increase. This difficulty is remedied with a countably infinite number of nonterminals. The grammar accounts for block size in the index of the nonterminals:

$$S_i \to q^i x S_i \mid t S_{i+1} \mid \epsilon \qquad i \ge 0. \tag{1.21}$$

If we start with S_0, the grammar has two quirks: it can generate an arbitrarily long string of x's before it produces any blocks of q's; and it can append an arbitrarily long string of t's to the end of a partition. Both of these quirks are helpful in solving the problem as originally posed. If we count all strings with j x's, then we will enumerate partitions with j or fewer parts. Likewise, if we examine strings with k t's then the maximum part size will be less than or equal to k.

The remaining challenge is to solve an infinite set of equations for S_0:

$$S_i = q^i x S_i + t S_{i+1} + 1 \qquad i \ge 0. \tag{1.22}$$

This is accomplished by applying the Q operator, $Qf(x) = f(qx)$, to equation (1.22) and noticing that we obtain the equation for S_{i+1}:

$$QS_i = q^{i+1} x Q S_i + tQ S_{i+1} + 1 \qquad i \ge 0. \tag{1.23}$$

So we expect that $QS_i = S_{i+1}$ in which case the infinite set of equations collapses to a single equation for S_0:

$$S_0 = xS_0 + tQ S_0 + 1. \tag{1.24}$$

More formally, we have the following theorem:

Fixed Point Theorem. *Let S_i be given by any system of equations of the form*

$$S_i = q^i O_1(S_i, S_{i+1}) + O_2(S_i, S_{i+1}) \qquad i \geq 0. \tag{1.25}$$

with operators O_1 and O_2 such that

$$QO_1(A, B) = qO_1(QA, QB) \qquad \text{and} \qquad QO_2(A, B) = O_2(QA, QB). \tag{1.26}$$

Let S^* be a solution of

$$S^* = O_1(S^*, QS^*) + O_2(S^*, QS^*), \tag{1.27}$$

then $S_i = Q^i S^*$ will satisfy the system (1.25).

Proof. *Apply Q^i to (1.27).*

The solution of (1.24) follows the pattern of binomial coefficients. We know that, without the Q operator (i.e., when $q = 1$),

$$\begin{aligned} S_0 &= xS_0 + tS_0 + 1 \\ S_0 &= \frac{1}{1 - x - t} \\ S_0 &= \sum_{j,k \geq 0} \binom{j+k}{j} x^j t^k. \end{aligned} \tag{1.28}$$

We claim that with the Q operator:

$$\begin{aligned} S_0 &= xS_0 + tQS_0 + 1 \\ S_0 &= \frac{1}{1 - x - tQ} \\ S_0 &= \sum_{j,k \geq 0} \binom{j+k}{j}_q x^j t^k, \end{aligned} \tag{1.29}$$

where the following definitions make the analogy work:

$$\begin{aligned} j!_q &= (1-q)(1-q^2)\dots(1-q^j)/(1-q)^j \\ \binom{j+k}{j}_q &= \frac{(1-q)(1-q^2)\dots(1-q^{j+k})}{(1-q)\dots(1-q^j)\,(1-q)\dots(1-q^k)} \\ \frac{1}{1-x-tQ} &= \sum_{m \geq 0} (x + tQ)^m. \end{aligned} \tag{1.30}$$

We assume, in the last sum, that the Q operators are applied to all factors on their right. Further properties of the Q operator and q-nomial coefficients can be found in [Andrews 1971].

Equations with the Q operator are not always as clean as the above example. Our next example uses the chain rule to extract useful information from a more difficult situation. It is a technique that appeared originally in the solution of problem 2.3.4.5–5 in [Knuth 1973], and will prove useful later in the analysis of algorithms.

Problem. *What is the average internal path length of an unlabelled tree? (Internal path length is the sum of the distances from each tree node to the root.)*

By modifying the grammar found at the beginning of this chapter we can preface each node of the tree with a block of q's. The length of the block of q's is equal to the depth of the node within the tree:

$$\begin{aligned} T_i &\to q^i\, x\, F_i \\ F_i &\to l\, T_{i+1}\, h\, F_i \mid \epsilon. \end{aligned} \qquad i \geq 0 \tag{1.31}$$

An application of the Q operator reveals that $QT_i = T_{i+1}$ so once again the infinite set of equations can be reduced to

$$\begin{aligned} T_0 &= xF_0 \\ F_0 &= (QT_0)F_0 + 1. \end{aligned} \tag{1.32}$$

This time the solution is not closed:

$$T_0 = \frac{x}{1 - QT_0}. \tag{1.33}$$

It can be expanded into a well known continued fraction of Ramanujan,

$$T_0 = \cfrac{x}{1 - \cfrac{qx}{1 - \cfrac{q^2x}{1 - \cfrac{q^3x}{1 - \cdots}}}}, \tag{1.34}$$

but for our purposes we do not need a complete solution, only an average path length. This suggests differentiating equation (1.33) with respect to q and setting q equal to 1. Let

$$U(x) = \left.\frac{\partial T_0}{\partial q}\right|_{q=1}. \tag{1.35}$$

We have already computed

$$V(x) = T_0|_{q=1} = \frac{1 - \sqrt{1 - 4x}}{2}. \tag{1.36}$$

Rewriting equation (1.33) and then differentiating the equation gives

$$\begin{aligned} T_0 - T_0QT_0 &= x \\ U - UV - V(V'x + U) &= 0, \end{aligned} \tag{1.37}$$

where the term in parentheses is the chain rule applied to QT_0. Algebraic manipulation yields:

$$\begin{aligned} U &= \frac{xVV'}{1-2V} \\ U &= \frac{x}{2}\,\frac{1-\sqrt{1-4x}}{1-4x} \\ U &= \sum_{n\geq 2}\frac{1}{2}\left(4^{n-1}-\binom{2(n-1)}{n-1}\right)x^n. \end{aligned} \tag{1.38}$$

For an average path length we divide by the total number of trees:

$$\frac{n\,2^{2n-3}}{\binom{2(n-1)}{n-1}}-\frac{n}{2}. \tag{1.39}$$

1.3 Differencing

The next example, taken from [Gross 1966], illustrates another technique for dealing with languages that do not have grammars in the usual sense. The problem is to enumerate Tutte triangulations.

Definition. *A Tutte triangulation is a division of a polygon into triangles such that no internal edges have both their endpoints on the boundary of the polygon. One of the external edges is marked with an orientation, as shown in Figure 1.4 below.*

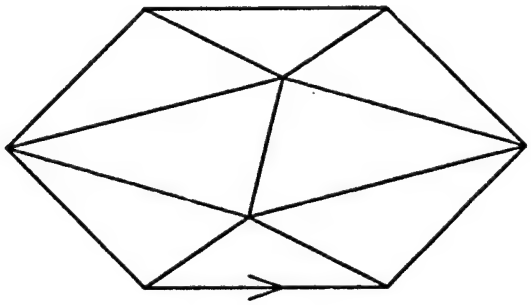


Figure 1.4 The Tutte Triangulation $c[tfc[tftftftft]ft]$.

A triangulation is assembled from individual triangles that are denoted by t's in the formal language. Initially each triangle has a marked, counterclockwise oriented edge. The structure is aggregated by two operations. The fanning operation, f, joins two structures by abutting the edge following the marked edge of the first structure with the marked edge of the second structure. The expression tft is shown in Figure 1.5 below. Notice that the marked edge of the first structure remains the marked edge of the whole structure.

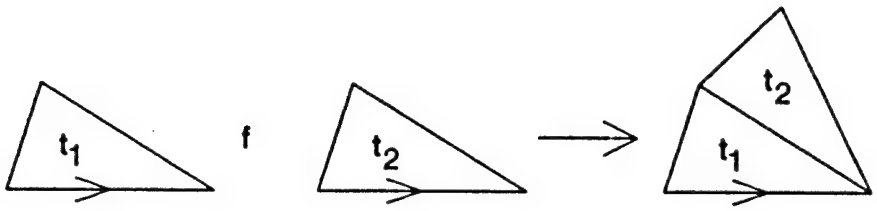


Figure 1.5 The Fanning Operation $t_1 f t_2$.

The closing operation, c, creates a new marked edge that is connected to the tail of the old marked edge and the far end of the edge following the marked edge. The closure of Figure 1.5, $c[tft]$, is pictured in Figure 1.6.

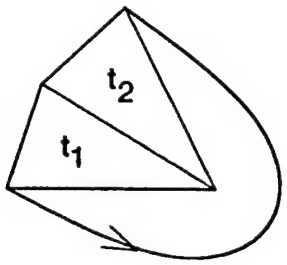


Figure 1.6 The Closing Operation $c[t_1 f t_2]$.

The grammar for Tutte triangulations specifies that the outermost, or last operation must be a closure, to prevent internal edges from bisecting the triangulation:

$$\begin{aligned} S &\rightarrow c[R] \mid t \\ R &\rightarrow S \mid R f S . \end{aligned} \tag{1.40}$$

If we strip off the outermost closure, what remains is a triangulation that allows bisecting edges to touch the head of the marked edge. These are represented by R in the above grammar; R can be decomposed by noting that R is either itself a Tutte triangulation, with no bisecting edges, or we can find the rightmost bisecting edge and split off a Tutte triangulation.

However, there is a serious flaw in (1.40). It makes no sense to close a single triangle, and in general, if a structure has j external edges we may apply c only $j-3$ times before an external triangle results. In order to avoid closing a triangle we would like to constrain the c's to be strictly less than the enclosed t's, yet this does not seem possible with a context-free grammar.

The following differencing trick allows us to derive a correct equation from the grammar at (1.40). Let T be all strings representing Tutte triangulations where the exterior polygon

is a triangle. Since S represents all Tutte triangulations, T is a subset of S, containing strings with one more t than c. If a string derived with (1.40) violates the constraint on c's and t's, then somewhere within the string a structure with triangular exterior is closed by a c operation. We prevent this from happening by subtracting the language $c[T]$ from the first nonterminal:

$$\begin{aligned} S &\to c[R] \mid t \quad - c[T] \\ R &\to S \mid R f S . \end{aligned} \tag{1.41}$$

Subtraction means that we eliminate a set of strings from the language derived from a nonterminal. If the strings eliminated are a proper subset, as is the case here, we can proceed with a translation:

$$\begin{aligned} S &= cR + t - cT \\ R &= \frac{S}{1 - fS} . \end{aligned} \tag{1.42}$$

These equations combine to give:

$$S + cT = t + \frac{cS}{1 - fS} . \tag{1.43}$$

The will always be one more t than f, so t can be replaced by 1. Now $T(f,c)$ is simple the diagonal terms of $S(f,c)$, and most features of interest are related to the occurrences of f and c, denoted respectively by $\mathcal{F}$ and C:

$$\begin{aligned} \text{Interior Triangles} &= \mathcal{F} + C + 1 \\ \text{Exterior Edges} &= \mathcal{F} - C + 3 \\ \text{Interior Edges} &= \mathcal{F} + 2C . \end{aligned} \tag{1.44}$$

Using a sophisticated application of Lagrange's theorem to equation (1.43), Tutte was able to find an expansion for S [Tutte 1962; pages 26–31]. The number of triangulations according to $\mathcal{F}$ and C turns out to be

$$\frac{2(4C+1)!}{(3C+2)!\,(C+1)!} \tag{1.45}$$

for $\mathcal{F} = C$, and

$$\frac{3(\mathcal{F}-C+2)!\,(\mathcal{F}-C-1)!}{(3\mathcal{F}+3)!} \sum_{j=0}^{\min(\mathcal{F}-C,C-1)} \frac{(3\mathcal{F}+C+1-j)!\,(\mathcal{F}-C+2+j)\,(\mathcal{F}-C-3j)}{j!\,(j+1)!\,(\mathcal{F}-C-j)!\,(\mathcal{F}-C+2-j)!\,(C-1-j)!} \tag{1.46}$$

for $\mathcal{F} > C$.

This completes our survey of the traditional uses of formal languages in enumeration and some of the techniques used to extend them beyond their context-free limitations. Additional examples of non-standard use can be found in the works of Cori [Cori 1970, 1972, and 1975], who uses them to study planar graphs, and in [Flajolet 1980], where they are used to count sequences of operations on data structures.

In the remaining chapters we will extend the usefulness of formal languages from unlabelled to labelled combinatorial objects. This is accomplished by adding partial orders that govern the labelling of terminal symbols, and that preserve the straightforward translation (of grammars into equations) seen throughout this chapter.

Chapter 2 introduces labelled formal languages, explains the relationship of grammars and partial orders, and demonstrates the wide range of classical generating functions that can be derived from this new framework. Chapters 3 and 4 explore some of the applications of labelled formal languages in diverse areas such as the analysis of algorithms and the counting and random generation of combinatorial objects.

The grammatical extensions that follow are believed to be new, although they bear some relation to recent efforts to vest analytic operations with combinatorial meaning (see for example [Joyal 1981]). The "diminished search trees" of Chapter 3 do not appear to be either known or analyzed by earlier authors, however the differential equations used in Chapter 3 are also used to study the closely related "median of n" modification of quicksort [Sedgewick 1975], [Knuth 1975]. Finally, the general purpose system described in Chapter 4 and implemented in appendix C, seems to be unique in its ability to generate a vast variety of combinatorial objects from short grammatical descriptions.

CHAPTER 2

LABELLED FORMAL LANGUAGES

In the previous chapter we found that grammars could express a variety of interesting structures and were easily translated into generating functions. However, the class of combinatorial objects expressible in this way is limited. We could encode unlabelled trees with l's and h's for the lower and higher movements of a tree walk:

$$xlxhlxh\,, \tag{2.1}$$

but it was not possible to generate labelled trees, such as

$$x_2lx_1hlx_3h \tag{2.2}$$

with ordinary grammars.

The purpose of this chapter is to define an extended form of grammar that will generate the second, labelled, string given above while retaining the nice translation property of the preceding chapter; it will still be possible to convert a grammar systematically into an equation and obtain a generating function, although the equations will typically be differential and the generating functions will always be exponential.

2.1 Smallest Label Control

Throughout this section the labelled tree problem will serve as a good illustration:

Problem. *How many rooted unordered labelled trees with n nodes are possible? [Cayley 1889] [Moon 1970]*

Unordered means that we do not care about the ordering of subtrees at each node. With ordered subtrees the answer is just $n!$ times the number of ordered unlabelled trees computed in Chapter 1, so it is the permuting of subtrees that makes this an interesting problem.

A labelled formal language has two new features. First, there is a special terminal character x. The occurrences of the special character receive distinct labels in the range 1 to n, where n is the total number of special characters. Second, every derived string, partial derivation and production in the grammar has an associated partial order that specifies acceptable ways of labelling the special characters. For example, suppose we derive the string

$$x_alx_blx_clh\,, \tag{2.3}$$

representing a small tree with two sons. Here a, b, and c are variables for the labels 1, 2, and 3 that have yet to be assigned. Since we do not care about the ordering of subtrees the two labellings

$$x_2 l x_1 h l x_3 h \qquad \text{and} \qquad x_2 l x_3 h l x_1 h \tag{2.4}$$

are redundant; only one should be produced by the grammar, so the partial order $b < c$ is associated with the string:

$$[b < c]\, x_a l x_b h l x_c h\,. \tag{2.5}$$

This way all labellings of the string will be in a canonical order, with the label of the first subtree less than the label of the second subtree.

We must now expect our grammar to produce both a string and a partial order. For this purpose variables are added to any nonterminal that can derive a string with x's:

$$\begin{aligned} T &\to [\,]\, x_a F_b \\ F &\to [e < f]\, l T_e h F_f \;\big|\; \epsilon\,. \end{aligned} \tag{2.6}$$

We also associate a partial order with each production possibility. In the production $T \to [\,]x_a F_b$, the b subscript on the nonterminal F is a variable for one of the labels 1 to n. However, since F can derive a string with many x's, the b stands for only the smallest label amongst the x's derived by F, if there is at least one x. Thus the production

$$F \to [e < f]\, l T_e h F_f \tag{2.7}$$

requires that the smallest label in the string derived from T_e be less than all the labels in the string derived from F_f. This constraint is precisely what we want for the tree problem, since this production is spinning off the descendants of a node in such a way that the smallest label will appear in the leftmost subtree, and of the remaining labels the smallest will appear in the second on the left, and so on. Notice that these smallest labels can appear anywhere within their subtrees; we are not ordering the roots of the subtrees as one might expect. Nevertheless, the final labelling, if it obeys the partial order fragments in each production, will be a canonical representation of a tree.

A grammar of an ordinary formal language is a set of rewrite rules. Beginning with the "start" nonterminal and repeatedly replacing nonterminals by one of their production possibilities eventually results in a string of all terminals. A similar process works for labelled formal languages. In the tree example we modify the grammar slightly by expanding the first production, so that

$$T \to [\,]\, x_a F_b \tag{2.8}$$

is changed to

$$T \to [a < b]\, x_a F_b \;\big|\; [c > d]\, x_c F_d\,. \tag{2.9}$$

This way each production possibility records the location of the smallest label.

The string portion of a labelled formal language functions like a rewrite system, with renaming of the subscript variables $a, b, c \ldots$ if necessary to insure that all the subscripts in any partial derivation are distinct.

The partial order is modified by substitution. Suppose we have a partial derivation,

$$[P]\, u S_a v\,, \tag{2.10}$$

and we apply the production

$$S \to [\mathcal{Q}]\, w, \tag{2.11}$$

where u, v and w are strings, $\mathcal{P}$ is a partial order on the subscripts of uS_av, and $\mathcal{Q}$ is a partial order on the subscripts of w. then S is rewritten to w, and a, in partial order $\mathcal{P}$, is replaced by the smallest item of partial order $\mathcal{Q}$. The remainder of $\mathcal{Q}$ is also added to $\mathcal{P}$, but with no additional relationships between $\mathcal{P}$ and $\mathcal{Q}$, save those implied by the addition of the smallest item of $\mathcal{Q}$ to $\mathcal{P}$. In brief, the application of (2.11) to (2.10) yields:

$$[\mathcal{P} \text{ with } \mathcal{Q} \text{ substituted for } a]\, uwv. \tag{2.12}$$

The substitution of partial orders is linked to the substitution of strings by the following requirement: if S is being rewriten to the empty string, then a must be maximal in $\mathcal{P}$. Or, equivalently, whenever a nonterminal has a subscript that is not maximal it must generate at least one x.

A labelled version of the small tree example should clarify the substitution process.

$$\begin{array}{ll} T \to [a < b]\, x_a F_b & (1) \\ T \to [c > d]\, x_c F_d & (2) \\ F \to [e < f]\, l T_e h F_f & (3) \\ F \to \epsilon & (4) \end{array} \tag{2.13}$$

The productions used are recorded on the right:

$$\begin{array}{ll} [\,]\, T & \\ & (2) \\ [c > d]\, x_c F_d & \\ & (3) \\ [c > e < f]\, x_c l T_e h F_f & \\ & (1) \\ [c > a < f; a < b]\, x_c l x_a F_b h F_f & \\ & (4) \\ [c > a < f]\, x_c l x_a h F_f & \\ & (3) \\ [c > a < g < i]\, x_c l x_a h l T_g h F_i & \\ & (1) \\ [c > a < j < i; j < k]\, x_c l x_a h l x_j F_k h F_i & \\ & (4) \\ [c > a < j < i]\, x_c l x_a h l x_j h F_i & \\ & (4) \\ [c > a < j]\, x_c l x_a h l x_j h & \end{array} \tag{2.14}$$

Note that the partial order obtained by this process is stronger than necessary since we only care that $a < j$ (compare with equation (2.5)). The extra strength is due to the production chosen in the first step of the derivation; using $T \to [a < b]\, x_a F_b$ would have derived another possibility, $[c < a < j]$. This is ultimately a consequence of our splitting the first production, $T \to [\,]\, x_a F_b$, into two cases $a < b$ and $b > a$. However, this splitting process is necessary, for in order to substitute one partial order into another we must know the smallest item in the substituted order.

The tree example illustrates another important constraint. At several points in the derivation the fourth production was applied and labels disappeared from the partial order.

At these points we cannot allow the label disappearing to be less than any other label. Thus the rewrite $F \rightarrow \epsilon$ can be preceded by $T \rightarrow [a < b]\, x_a F_b$, but not by $T \rightarrow [c > d]\, x_c F_d$. This requirement eliminates unwanted ambiguity.

In summary, an labelled formal language describes two intimately linked actions: a rewriting process for strings and a substitution process for partial orders. As the grammar rewrites a string it weaves together a partial order that specifies acceptable labellings of the special characters.

2.2 Translation to Generating Functions

The simplest and perhaps most useful labelled formal languages have productions of the form

$$R \rightarrow [b < a; b < c; b < d \ldots]\, S_a T_b U_c V_d \ldots . \tag{2.15}$$

where the partial order specifies one label, in this case b, to be less than all the others. For the time being we will restrict our attention to these simple partial orders, and use the following shorthand notation: a box superscript marks the nonterminal or terminal receiving the smallest label, so the above production would be noted as:

$$R \rightarrow ST^{\square}UV \ldots , \tag{2.16}$$

and the tree grammar of the preceding section would appear as: .

$$\begin{aligned} T &\rightarrow xF \\ F &\rightarrow lT^{\square}hF \mid \epsilon . \end{aligned} \tag{2.17}$$

The absence of a box on the first tree production denotes the absence of constraints. We could also have written,

$$T \rightarrow x^{\square}F \mid xF^{\square} , \tag{2.18}$$

expanding the production as we did in the preceding section. In later sections we will explore more complex partial orders.

The translation of labelled formal languages to generating functions proceeds as follows:
1) Convert the grammar to a set of equations by changing

a) $\rightarrow$ to $=$
b) $\mid$ to $+$
c) ϵ to 1
d) $ST^{\square}UV \ldots$ to $\int(ST'UV \ldots)$

2) Solve the differential equations for the start symbol and treat the result as a multivariate generating function that is exponential in the special character, and ordinary in the other terminal symbols.

The tree grammar, for example, translates to

$$\begin{aligned} T &= xF \\ F &= \int (lT'hF) + 1 . \end{aligned} \tag{2.19}$$

The l's and h's are not particularly interesting to count, so l and h are set to 1. After differentiation the second equation becomes

$$F' = T'F, \tag{2.20}$$

with solution $F = e^T$, so we obtain the classic, implicit, generating function for labelled trees [Polya 1937]:

$$T = xe^T. \tag{2.21}$$

Notice that it doesn't matter which form of the first production we use for the translation. The first version $T \to xF$, yields $T = xF$ while the second version, $T \to x^{\square}F \mid xF^{\square}$, yields $T = \int x'F + \int xF'$, but both are equivalent under integration by parts. Combinatorially, the integration by parts rule simply states that the smallest label must appear in one of the two substructures on the right hand side.

Why does this translation work? The key step is 1.d, where $ST^{\square}UV\ldots$ is changed to $\int(ST'UV\ldots)$. The other aspects of the translation are similar to the procedures used for ordinary formal languages. Step 1.d, however, integrates the whole term, with the derivative placed on the boxed terminal or nonterminal within the term. Both the integration and the differentiation are with respect to the special character, in this case x, which is now treated as a commutative variable. The constant of integration is always zero, that is, $\int f(x)$ means $\int_0^x f(y)dy$.

It is not hard to see why this works. Suppose we have a production $R \to ST$ which translates to $R = ST$, and we have two exponential generating functions for S and T:

$$\begin{aligned} S &= \sum_{i\geq 0} s_i \frac{x^i}{i!} \\ T &= \sum_{j\geq 0} t_j \frac{x^j}{j!}. \end{aligned} \tag{2.22}$$

Then the product is given by

$$R = \sum_{k\geq 0}\left(\sum_{i\geq 0}\binom{k}{i} s_i t_{k-i}\right)\frac{x^k}{k!}, \tag{2.23}$$

where the inner term, $\binom{k}{i}s_it_{k-i}$, builds a labelled derivation for R by taking any derivation s_i for S, combined with any derivation t_{k-i} for T, and relabelling so that the relative orders of the labels within S and T individually are maintained, but the two sets of labels are intermingled in all possible ways, $\binom{k}{i}$.

Suppose now that we insist that the smallest label appear in the subtree derived from T by writing $R \to ST^{\square}$. With exponential generating functions, the integration and differentiation in the translation of this production, $R = \int(ST')$, act like shift operators:

$$T' = \sum_{j\geq 0} t_{j+1}\frac{x^j}{j!}. \tag{2.24}$$

Therefore the product,

$$R = \sum_{k\geq 0}\left(\sum_{i\geq 0}\binom{k-1}{i}s_i t_{k-i}\right)\frac{x^k}{k!}, \tag{2.25}$$

corresponds to shifting the smallest label of T away, intermingling the remaining labels, and then returning the smallest label. We obtain $\binom{k-1}{i}$ rather than $\binom{k}{i}$.

2.3 Examples

This section reviews a variety of classic enumeration problems in order to show the broad applicability of labelled formal languages. For each problem we will find an encoding, grammar, translation, and solution. Since some of the solutions will be given implicitly, by equations like $T = xe^T$, a special appendix has been devoted to the techniques of Lagrange and Bell for recovering meaningful information from such equations. For each of the classic problems two references are given, the first to the original source of the problem, and the second to a more accessible modern reference.

2.3.1 Alternating Permutations

Problem. *Count the number of permutations $\sigma_1\sigma_2\ldots\sigma_n$ of $1,2,\ldots,n$ that obey $\sigma_{2i-1} > \sigma_{2i} < \sigma_{2i+1}$ for all i. In these permutations the first entry is large and thereafter large and small strictly alternate. [André 1878] [Comtet 1974; p. 258]*

For the sake of clarity we restrict the problem to odd length permutations. The encoding of these permutations is simply a string of x's, with the ith x having label σ_i. So for $n = 3$ there are two correctly alternating permutations, $x_2x_1x_3$ and $x_3x_1x_2$.

The grammar for alternating permutations,

$$A \to Ax^{\square}A \mid x, \tag{2.26}$$

is based on the observation that the smallest labelled x in the permutation splits the permutation into two correctly alternating pieces. The translation of the grammar,

$$A = \int A^2 + x, \tag{2.27}$$

reduces to a differential equation

$$A' = A^2 + 1 \tag{2.28}$$

with solution $A = \tan x$.

This example raises some interesting questions of ambiguity. In the preceding chapter a grammar had to be unambiguous to be useful for enumeration. The same holds for labelled formal languages, although in this case the underlying ordinary grammar $A \to AxA \mid x$ is very ambiguous. It is the box operator, the augmentation to the grammar, that makes this an unambiguous grammar and allows us to obtain a useful generating function.

2.3.2 Stirling Numbers of the First Kind

Problem. *How many permutations of n elements have exactly k cycles?*

This time permutations are encoded by their cycle structure. For example, the permutation taking 1234 to 3241 has cycle structure (2)(314), but since (2)(143) and (314)(2) also represent the same permutation we will disambiguate these possibilities by writing cycles with their smallest element first, and arranging the cycles so that the minimums increase from left to right, e.g., (143)(2). This is captured with the grammar

$$\begin{aligned} P &\to C^{\square} bP \mid \epsilon \\ C &\to x^{\square} R \\ R &\to xR \mid \epsilon . \end{aligned} \tag{2.29}$$

The first production lays out the cycle structure, with b's separating the cycles. The box operator insures that the cycles will have successively larger minimum elements. The last two productions derive a cycle: $C \to x^{\square} R$ makes the first item the smallest in the cycle and $R \to xR \mid \epsilon$ finishes the cycle without constraint.

The translation and solution of this grammar is straightforward,

$$\begin{aligned} R &= \frac{1}{1-x} \\ C &= \int \frac{1}{1-x} = -\ln(1-x) \\ P &= \int (C' bP) + 1 \\ P' &= C' bP \\ P &= e^{-b\ln(1-x)} , \end{aligned} \tag{2.30}$$

giving the familiar generating function for Stirling numbers of the first kind:

$$e^{-b\ln(1-x)} = (1-x)^{-b} = \sum_{n,j} \begin{bmatrix} n \\ j \end{bmatrix} b^j \frac{x^n}{n!} . \tag{2.31}$$

Two modifications of the above grammar result in other well known generating functions. By insisting that no cycle contains a single element,

$$\begin{aligned} P &\to C^{\square} bP \mid \epsilon \\ C &\to x^{\square} xR \\ R &\to xR \mid \epsilon , \end{aligned} \tag{2.32}$$

we obtain derangements (permutations without fixed elements). This grammar converts to

$$P = \left(\frac{e^{-x}}{1-x} \right)^b . \tag{2.33}$$

By insisting that each cycle contains no more than two elements,

$$\begin{aligned} P &\to C^{\square} bP \mid \epsilon \\ C &\to x^{\square} x \mid x, \end{aligned} \tag{2.34}$$

we obtain involutions (permutations σ such that $\sigma^2(x) = x$) with a generating function

$$P = e^{-b(x+x^2/2)}. \tag{2.35}$$

2.3.3 Stirling Numbers of the Second Kind

Problem. *Count the number of partitions of 1 to n into j nonempty subsets. This corresponds to placing n numbered balls into j indistinguishable boxes, disregarding the ordering of balls within boxes.*

The partition is encoded as a series of labelled x's, with p's marking the boundary between subsets. Within a subset, since the order doesn't matter, the x's are arranged in ascending order of their labels. Thus there are three partitions of $n = 3$ into $j = 2$ parts:

$$\begin{aligned} &x_1 p x_2 x_3 p \\ &x_1 x_2 p x_3 p \\ &x_1 x_3 p x_2 p \end{aligned} \tag{2.36}$$

An extra p appears at the right of each string so that there are as many p's as blocks in the partition.

The grammar,

$$\begin{aligned} F &\to B^{\square} pF \mid \epsilon \\ B &\to x^{\square} R \\ R &\to x^{\square} R \mid \epsilon, \end{aligned} \tag{2.37}$$

functions by first laying out the blocks of the partition arranged in increasing order of their smallest labels. Inside a block, the last two productions arrange the labels in ascending order, and insure that there is at least one x per block.

Beginning with the last production we can transform and solve the grammar:

$$\begin{aligned} R &= \int R + 1 \\ R &= e^x \\ B &= \int R \\ B &= e^x - 1 \\ F &= p \int B' F + 1 \\ F &= e^{p(e^x - 1)}. \end{aligned} \tag{2.38}$$

Notice that when e^z is integrated to find B, we must subtract 1 to make the constant of integration zero. The solution gives a generating function for the Stirling numbers of the second kind,

$$e^{p(e^z-1)} = \sum_{n,j} \left\{ n \atop j \right\} p^j \frac{x^n}{n!}. \tag{2.39}$$

Later on, we will find use for the associated Stirling numbers of the second kind, denoted by $\left\{ n \atop j \right\}_{/s}$, and equal to the number of ways of partitioning n into j subsets with at least s items per subset. Rewriting the above grammar to force larger subsets,

$$\begin{aligned} F &\rightarrow B_s^{\square} pF \mid \epsilon \\ B_i &\rightarrow x^{\square} B_{i-1} \qquad 1 \le i \le s \\ B_0 &\rightarrow x^{\square} B_0 \mid \epsilon, \end{aligned} \tag{2.40}$$

and translating this grammar, gives a generating function for associated Stirling numbers:

$$F = e^{p\left(e^z - \sum_{0 \le i \le s} \frac{x^i}{i!}\right)}. \tag{2.41}$$

2.3.4 Mappings

A mapping of the integers 1 to n into itself will consist of several disjoint components, each with a central cycle. Pictorially, a mapping looks something like this:

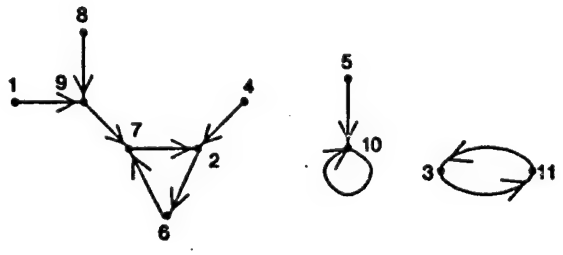


Figure 2.1 A Mapping of 1 to 11 into the Same Range

Problem. *How many mappings $f : \{1 \ldots n\} \rightarrow \{1 \ldots n\}$ have exactly j components?*

If a mapping has only one loop and no other cycles then the mapping is a labelled tree. On the other hand, a surjection will consist entirely of cycles, with no tree-like structure.

The grammar for mappings combines these two extremes; it includes a both a tree generator and permutation generator:

$$\begin{aligned} P &\to C^{\square} bP \mid \epsilon \\ C &\to T^{\square} R \\ R &\to TR \mid \epsilon \\ T &\to xF \\ F &\to lT^{\square} hF \mid \epsilon . \end{aligned} \tag{2.42}$$

The first three productions have already appeared in Section 2.3.2 on permutations and Stirling numbers of the first kind, and the last two productions form a familiar tree generator. Together these productions generate mappings, coded with b's separating components of the mapping, and lists of trees within components ordered so that the tree roots form the cycles. The mapping pictured above would encode as:

$$x_7 l x_9 l x_1 h l x_8 h h x_2 l x_4 h x_6 b x_3 x_{11} b x_{10} l x_5 h b . \tag{2.43}$$

Portions of the solution of this grammar are given implicitly,

$$\begin{aligned} F &= e^T \\ T &= xe^T \\ R &= \frac{1}{1-T} \\ C &= -\ln(1-T) \\ P &= e^{-b\ln(1-T)} , \end{aligned} \tag{2.44}$$

so we don't have a closed form for P. However, using the techniques of Appendix A, detailed information can be recovered from the generating function:

$$P = \sum_{n,j} \left(\sum_k \binom{n-1}{k-1} n^{n-k} \begin{bmatrix} k \\ j \end{bmatrix} \right) b^j \frac{x^n}{n!} . \tag{2.45}$$

Metropolis and Ulam initiated the study of the number of components in a random mapping with some empirical results [Metropolis 1953]. Subsequent authors were able to compute the expected number of components and give complex formulas for the distribution. The comparatively tight expression in (2.45) was discovered by Riordan [Riordan 1962]. Other questions about random mappings such as the number of recurrent elements (those involved in cycles) have been studied and are also amenable to treatment with labelled formal languages.

2.3.5 Schröder's Third Problem

Problem. *A set of n labelled elements is chained together in groups of size m as follows: we repeatedly collect m elements, delete them from the set and then add them back together as a single, new, grouped element until only one element remains in the set. This single element is the root of a labelled m-way branching tree. How many such trees or chainings are there? [Schröder 1870] [Comtet 1974; p. 165]*

The problem is solved with a modified tree grammar, designed to force exactly m descendants at each node:

$$\begin{aligned} T &\to bS_m e \mid x \\ S_i &\to T^{\square} S_{i-1} \qquad 1 < i \le m \\ S_1 &\to T . \end{aligned} \tag{2.46}$$

A matched pair b (begin) and e (end) mark the left and right ends of chains. So, for example, there are ten chains of $n = 5$ elements with grouping factor $m = 3$:

$$\begin{array}{ll} bbx_1x_2x_3ex_4x_5e & bbx_1x_4x_5ex_2x_3e \\ bbx_1x_2x_4ex_3x_5e & bx_1bx_2x_3x_4ex_5e \\ bbx_1x_2x_5ex_3x_4e & bx_1bx_2x_3x_5ex_4e \\ bbx_1x_3x_4ex_2x_5e & bx_1bx_2x_4x_5ex_3e \\ bbx_1x_3x_5ex_2x_4e & bx_1x_2bx_3x_4x_5ee \end{array} \tag{2.47}$$

Notice that since the problem specifies no order among the m terms in a chain, the grammar uses a box operator to specify a canonical left to right, smallest to largest layout.

Dropping b and e from the problem, the grammar translates to:

$$\begin{aligned} T &= S_m + x \\ S_i &= \int T' S_{i-1} \qquad 1 < i \le m \\ S_1 &= T , \end{aligned} \tag{2.48}$$

from which we conclude that

$$S_i = \frac{T^i}{i!} \tag{2.49}$$

and

$$T = x + \frac{T^m}{m!} . \tag{2.50}$$

Once again the generating function is given implicitly and the techniques of Appendix A are applicable:

$$\left\langle \frac{x^n}{n!} \right\rangle T = (n-1)! \binom{-n}{(n-1)/(m-1)} \left(\frac{-1}{m!} \right)^{(n-1)/(m-1)} , \tag{2.51}$$

when $m - 1$ divides $n - 1$.

2.3.6 Schröder's Fourth Problem and Series-Parallel Networks

Problem. *A Schröder system is a collection of subsets of the integers 1 to n that includes each singleton subset $\{i\}$, the whole set, $\{1,2,3,\ldots\}$, and other subsets that are strictly hierarchical, that is, $A \subset B$, $B \subset A$, or $A \cap B = \emptyset$. We wish to count the number of Schröder systems of n elements. [Schröder 1870] [Comtet 74; p. 224]*

Since they are hierarchical, Schröder systems can be generated with labelled formal languages. The terminals b and e are used to bracket each subset:

$$\begin{aligned} S &\to bxe \mid bS^{\square}Me \\ M &\to S^{\square}Q \\ Q &\to S^{\square}Q \mid \epsilon . \end{aligned} \tag{2.52}$$

Notice that $S \to bxe$ encloses every integer in its own subset. The combination of productions $S \to bS^{\square}Me$ and $M \to S^{\square}Q$ eliminates redundant subsets like $bbbx_1eee$.

Working backwards through the grammar, the productions convert to equations that are readily solved:

$$\begin{aligned} Q &= \int S'Q + 1 \\ Q &= e^S \\ M &= \int S'Q \\ M &= e^S - 1 \\ S &= bex + be \int S'M \\ S &= bex + be\left(e^S - S - 1\right) . \end{aligned} \tag{2.53}$$

For counting purposes b and e are redundant—one of them can be dropped from the equation. Applying Lagrange inversion to

$$S = bx + b\left(e^S - S - 1\right) \tag{2.54}$$

gives an expansion in terms of the associated Stirling numbers of the second kind:

$$S = \sum_{j,n} \left\{ {n+j-1 \atop j} \right\}_{/2} b^{n+j} \frac{x^n}{n!} . \tag{2.55}$$

Networks of resistors motivate a problem that is closely related to Schröder systems. Every resistor has a different label, and the ordering within a series or parallel group is irrelevant. So, for instance, there are only eight significantly different networks of three labelled resistors:

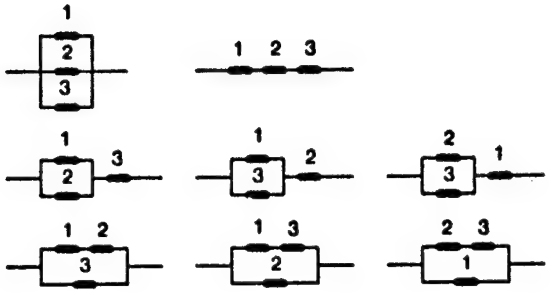


Figure 2.2 All Eight Networks of Three Resistors

Problem. *Count the number of distinct networks of n resistors.*

A network is encoded by using p and q (backwards p) to bracket a set of circuits in parallel, and s and z to analogously bracket circuits in series. The string between a matched p and q can contain individual resistors, x_i, or series circuits enclosed in s and z. So the above eight networks of three resistors would encode as:

$$\begin{array}{lll} px_1x_2x_3q & sx_1x_2x_3z & \\ spx_1x_2qx_3z & spx_1x_3qx_2z & sx_1px_2x_3qz \\ psx_1x_2zx_3q & psx_1x_3zx_2q & px_1sx_2x_3zq \end{array} \tag{2.56}$$

Needless to say, there is a great deal of symmetry between series and parallel. The three productions that generate a parallel circuit,

$$\begin{aligned} A &\to x \,|\, pE^{\square}B \\ B &\to E^{\square}C \\ C &\to E^{\square}C \,|\, q, \end{aligned} \tag{2.57}$$

are identical to the three productions for series circuits,

$$\begin{aligned} E &\to x \,|\, sA^{\square}F \\ F &\to A^{\square}G \\ G &\to A^{\square}G \,|\, z. \end{aligned} \tag{2.58}$$

To this we might add an initial production,

$$T \to A \,|\, E, \tag{2.59}$$

that allows the whole network to be series or parallel, but unfortunately this production generates two different single resistor networks; everything is fine, except for the case $n = 1$. To remedy this, we advance the first production beyond A and E,

$$T \to x \,|\, pE^{\square}B \,|\, sA^{\square}F, \tag{2.60}$$

and include only one copy of x.

When translating this grammar, A can be expressed in terms of E,

$$A = x + p\left(e^E - E - 1\right), \tag{2.61}$$

and likewise, E can be expressed in terms of A,

$$E = x + s\left(e^A - A - 1\right). \tag{2.62}$$

Since we are seeking the total number of circuits of n resistors, independent of the number of internal series and parallel constructs, we need only invert $E = x + e^E - E - 1$ to obtain

$$E = \left(\sum_{0\le j<n} \left\{ {n+j-1 \atop j} \right\}_{/2}\right) \frac{x^n}{n!}. \tag{2.63}$$

Except for the case $n = 1$ the number of series-parallel networks is twice this coefficient,

$$2\left(\sum_{0\le j<n} \left\{ {n+j-1 \atop j} \right\}_{/2}\right). \tag{2.64}$$

The connection with Schröder systems should now be apparent. A Schröder system becomes a series-parallel network when it is "striped," by choosing a series or parallel nature for the largest set, and then alternating series and parallel down the chains of successively smaller subsets.

The study of networks was begun by MacMahon [MacMahon 1892], who gave a formula for the unlabelled problem. Knödel found the implicit generating function for the labelled problem [Knödel 1951] and Carlitz and Riordan noticed the correspondence with Schröder's problem [Carlitz 1959]. A good exposition, combining the labelled and unlabelled variations, can be found in [Riordan 1978].

2.3.7 Eulerian Numbers

Problem. *A descent in a permutation is a pair of adjacent elements such that $\sigma_i > \sigma_{i+1}$. We wish to count the number of permutations with j descents.*

For this problem we encode the permutation directly in the labels of the x's. A g is inserted between every pair of x's with decreasing labels. For example, there are four permutations of $n = 3$ elements with $j = 1$ descent:

$$\begin{array}{ll} x_1x_3gx_2 & x_2gx_1x_3 \\ x_2x_3gx_1 & x_3gx_1x_2 \end{array} \tag{2.65}$$

The grammar for this encoding relies on a simple fact: the smallest label will always cause a descent, unless it is at the left end of the permutation:

$$E \to Egx^{\square}E \;\big|\; x^{\square}E \;\big|\; Egx^{\square} \;\big|\; x. \tag{2.66}$$

The differential equation derived from this grammar,

$$E' = gE^2 + E + gE + 1, \tag{2.67}$$

is complicated by the nonlinear function of E on the right side of the equation. Since this type of equation has appeared before (with alternating permutations, Subsection 2.3.1), we pause now to consider a solution strategy.

If the right side of the equation has no constant term,

$$Y' = k_1 Y^2 + k_2 Y, \tag{2.68}$$

then the transformation $Y = 1/Z$ yields a first order linear differential equation,

$$-Z' = k_1 + k_2 Z. \tag{2.69}$$

When a constant term is present in the original problem,

$$Y' = k_1 Y^2 + k_2 Y + k_3, \tag{2.70}$$

then a constant c is also added to the transformation,

$$Y = \frac{1}{Z} + c, \tag{2.71}$$

giving an equation

$$-Z' = k_1(1 + cZ)^2 + k_2(1 + cZ)Z + k_3 Z^2. \tag{2.72}$$

First $k_1 c^2 + k_2 c + k_3 = 0$ is solved for c to eliminate the Z^2 term from the right side of the equation, and to reduce the problem to a first order linear differential equation.

Returning to the problem of descents, we find

$$E = \frac{1}{Z} + c_1 \tag{2.73}$$

$$gc_1^2 + (g+1)c_1 + 1 = 0 \tag{2.74}$$

$$c_1 = -1 \quad \text{or} \quad -\frac{1}{g} \tag{2.75}$$

(Curiously, the choice of root here does not matter, so $c_1 = -1$ is used.)

$$\begin{aligned} -Z' &= g - 2gZ + (g+1)Z \\ Z' &= (g-1)Z - g \\ Z &= \frac{g}{g-1} + c_2 e^{x(g-1)} \end{aligned} \tag{2.76}$$

Since the grammar does not generate an empty string, $E(0)$ is zero, $Z(0)$ is one and so $c_2 = -1/(g-1)$. Assembling the results, we obtain the classic generating function for Eulerian numbers [Euler 1755; p. 487]:

$$E = \frac{g-1}{g - e^{x(g-1)}} - 1. \tag{2.77}$$

A recurring theme of the preceding examples is the idea of laying out a structure in canonical form. A group acts on some portion of the structure, and yet judicious use of the box operator insures that only one representative from each equivalence class is generated by the grammar and counted by the generating function. So far we've seen the symmetric group (acting on such things as the descendants of nodes in trees, and collections of indistinguishable subsets) and the cyclic group (acting on the cycles within permutations). To these two groups we can add two closely related groups, the alternating group and the dihedral group. The use of the box operator with these four well known groups is summarized below. The elements of the group are denoted by E's which can be either single x's or nonterminals that derive structures containing x's.

1) The Symmetric Group, S_n:

$$S \to E^{\Box} S \mid \epsilon. \tag{2.78}$$

Here the elements of the group are arranged in sorted order, based on the smallest label within each element.

2) The Alternating Group, A_n. The grammar must permit an additional degree of freedom in the last two elements:

$$S \to E^{\Box} S \mid EE. \tag{2.79}$$

3) The Cyclic Group, C_n. A cycle is in canonical form when the smallest label is in the first element,

$$\begin{aligned} S &\to E^{\Box} T \\ T &\to ET \mid \epsilon. \end{aligned} \tag{2.80}$$

4) The Dihedral Group, D_n. The dihedral group is laid out in the order depicted below:

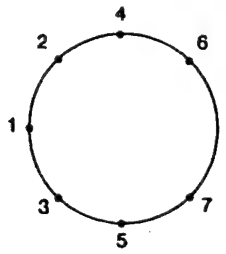


Figure 2.3 Layout for the Dihedral Group

where the group is generated by a cyclic shift, $(1357\ldots642)$, and a flip, $(1)(23)(45)(67)\ldots$. The grammar places the smallest label in the first element and insists that the second element contains a smaller label than the third:

$$\begin{aligned} S &\to E^{\Box} TU \\ T &\to E^{\Box} E \\ U &\to EU \mid \epsilon. \end{aligned} \tag{2.81}$$

In this way the four common groups can be encoded with labelled formal languages.

2.4 A Generalized Definition

From a bottom-up perspective, the labelled production $C \rightarrow AB$ constructs a string C by first constructing two labelled strings, A and B; then concatenating the strings; and then shuffling the labels together, in much the same way that two decks of cards are shuffled together. The presence of a box operator, $C \rightarrow A^{\Box}B$ constrains the shuffle so that the smallest label in A is also the smallest label in C; one card is flipped down from the A deck at the beginning of the shuffle.

Pursuing the card analogy further, a "good" dealer can control more than just the first card on the bottom of the deck. The intermingling of the first few cards as well as the last few cards can be manipulated by the dealer as he shuffles. In the definitions that follow, two partial orders are introduced in the grammars. The partial order enclosed in brackets constrains the mingling of the smallest labels while the partial order in braces controls the largest labels. So, for instance, the production

$$C \rightarrow \{e > f > g\}\,[b < c < a]\,A^{\{ef\}}_{[a]}\,B^{\{g\}}_{[bc]} \tag{2.82}$$

specifies that the two largest labels will appear in the string derived from A and the two smallest labels will be in B's derivation.

Before we attack the general definition of labelled formal languages we need some basic facts about partial orders. It is helpful to have a definition of partial orders that distinguishes between relations and sets:

Definition. *A poset is a set S together with a relation $\mathcal{S}$ such that:*

1) $a\mathcal{S}b \Rightarrow \neg b\mathcal{S}a$ (antisymmetric)

2) $a\mathcal{S}b \wedge b\mathcal{S}c \Rightarrow a\mathcal{S}c$ (transitive)

$\mathcal{S}$ is usually described as a series of inequalities amongst the elements of S, such as $a < b$, $b < c$. The transitive closure of these inequalities is $\mathcal{S}$ itself.

Definition. *A linear embedding of a partial order is a mapping m of S into the integers such that $a\mathcal{S}b \Rightarrow m(a) < m(b)$.*

We need to carefully separate those labels that will be arranged according to a partial order and those labels that will be "shuffled" in a less constrained way. For this purpose we introduce two new notions: *active* elements and *boundary* elements. *Active* elements will be counted with a partial order, while boundary elements will be "shuffled." They may appear in a partial order, but it is only to mark the edge of the active elements. For the smallest labels we have the following definition:

Definition. *An element β is a boundary element if there are no elements $\alpha > \beta$.*

(In the definitions that follow, the reader can supply analogous definitions for the largest labels.) For a production like

$$S \rightarrow [b < c < a]\,A_{[a]}B_{[bc]}\,, \tag{2.83}$$

a is a boundary element while b and c are not. Since the subscripts on a nonterminal represent the smallest few labels we could extend these subscripts through more elements.

In the last production we could write $B_{[bcd]}$ with $b < c < d$. In this case d is also a boundary element, but to avoid adding d to the subscripts of B we will usually write something like equation (2.83) and say that B has an implicit boundary element.

Definition. *An element is active if it is less than every explicit or implicit boundary element.*

Thus, in the example above, b and c are active elements.

Definition. *A partial order is well separated if all elements are either active or boundary.*

Again, our example above is *well separated*. *Well separateness* is crucial to the translation of grammars into integral equations. We will use differentiation and integration to remove active elements and count them according to partial orders.

For purposes of derivation however, we need to be able to substitute partial orders. When a string contains a nonterminal like $B_{[bc]}$, and we wish to rewrite the nonterminal, we must identify the two smallest elements in the derivation of B. This is possible if B has a production $B \to [\mathcal{A}] \dots$ where $\mathcal{A}$ is 2-smallest determined:

Definition. *A k-smallest determined poset has k elements, $a_1, a_2, a_3, \dots, a_k$ such that $a_1 < a_2 < a_3 \cdots < a_k$ and all other elements in S are greater than a_k.*

For purposes of the definition we include any implicit boundary points. This insures that $a_1, a_2, \dots, a_k$ will be less than all labels, even those not involved in the partial orders.

Definition. *(Substitution of Partial Orders) Let $\mathcal{Q}$ be a k smallest determined partial order over a set Q with k smallest elements $q_1 < q_2 < q_3 < \cdots < q_k$. Let $\mathcal{P}$ be a partial order over P with k (not necessarily smallest) elements $p_1 < p_2 < p_3 < \cdots < p_k$. Then the substitution of $\mathcal{Q}$ for $p_1, p_2, p_3, \dots, p_k$ in $\mathcal{P}$ is a partial order $\mathcal{R}$ over $(P - \{p_1, p_2, \dots, p_k\}) \cup Q$ consisting of $\mathcal{P}$, $\mathcal{Q}$, and some extra patching relations:*

1) $p_i < p \in \mathcal{P} \Rightarrow q_i < p \in \mathcal{R}$

2) $p_i > p \in \mathcal{P} \Rightarrow q_i > p \in \mathcal{R}$ (q_i acts like p_i)

3) $p < p_i \in \mathcal{P} \wedge q_i < q \in \mathcal{Q} \Rightarrow p < q \in \mathcal{R}$ (transitive closure across the identification $q_i = p_i$)

Postponing for the moment the definition of label-controlled strings and derivation steps, we can define ...

Definition. *A labelled grammar is a five-tuple (T, x, N, S, P) consisting of*

1) a terminal alphabet T,

2) a special symbol $x \in T$,

3) a nonterminal alphabet N,

4) a start symbol $S \in N$,

5) a set of productions P of the form

$$C \rightarrow \text{a label-c}\ldots\text{olled string}\ , \tag{2.84}$$

where $C \in N$.

A label-controlled string D is derived by the grammar if it is possible to begin with S and obtain D after a finite number of derivation steps.

Definition. *A label-controlled string is a string over the same alphabet as the formal language (N, x, T) where each nonterminal has a (possibly empty) collection of superscripts and subscripts. The string is prefixed with two partial orders, one over the superscripts (enclosed in braces) and the other over the subscripts (enclosed in brackets). The occurrences of the special character x in the string have at most one superscript and one subscript. If two scripts are present, then one must be a boundary element.*

By insisting that at least one label of a doubly labelled x be boundary, the last definition prevents the two partial orders from interfering with one another.

Definition. *(Derivation Step.) The label-controlled string*

$$\{P\}[A]\, wF^{\{p_1 p_2 \ldots p_j\}}_{[a_1 a_2 \ldots a_k]} v \tag{2.85}$$

derives in one step $\{R\}[C]\, wGv$ if:

1) There is a production $F \rightarrow \{Q\}[B]\, G$. where Q and B are partial orders on the superscripts and subscripts of G.

2) Either

a) G has no x's, $j \leq 1$, $k \leq 1$, and p_1, a_1 are boundary elements of P and A. R is then the result of deleting p_1 from P, and C is the result of deleting a_1 from A.

b) Q is j largest determined and R is the result of substituting Q for $p_1, p_2, \ldots, p_j$ in P. B is k smallest determined and C is the result of substituting B for $a_1, a_2, \ldots, a_k$ in A.

As the definition is writen, possibility 2.a is the only way that labels can disappear from partial orders. This is adequate for the definition, but we will sometimes find it convenient to have F derive a string of all terminals several of which are special characters. The essential point is that only boundary elements can disappear from partial orders.

Definition. *A full derivation is a mapping of the labels of a label-controlled string (containing only terminal symbols) onto the integers 1 to n. The mapping must be a linear embedding of both of the partial orders associated with the string.*

We turn now to the transformation of labelled grammars into integral equations, for which we will need to identify the active elements in the partial orders.

Definition. *The translation of a string, $\{\mathcal{P}\}[\mathcal{A}]w$, with well separated posets $\mathcal{P}$ and $\mathcal{A}$, is the product of*

1) The number of linear embeddings of the active elements of $\mathcal{P}$.

2) The number of linear embeddings of the active elements of $\mathcal{A}$.

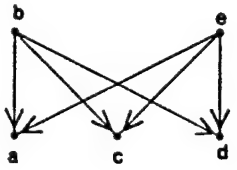
3) The integral repeated as many times as there are active elements in $\mathcal{P}$ and $\mathcal{A}$, of the product of the symbols in w, each differentiated as many times as there are active elements in their subscripts and superscripts.

Definition. *The translation of a grammar is obtained by summing for every nonterminal $V \in N$ the translation of each production possibility for V, and setting this sum equal to V.*

For example, the production

$$T \to [\mathcal{A}]\, S_{[ab]} x_{[c]} S_{[de]}\,, \tag{2.86}$$

with partial order

$\mathcal{A} =$ 

(2.87)

has two boundary elements, b and e, and three active elements, a, c, and d, so the production possibility is well separated. The translation is

$$T = 6 \iiint (S')^2\,. \tag{2.88}$$

The above definitions make two, somewhat different, demands on the partial orders used in the grammars. A derivation step requires that the partial orders be k-smallest or largest determined in order to make substitutions. On the other hand, the process of translation to integral equations requires that the partial orders be *well separated*. In fact, the original grammar can be presented in a way that meets neither of these requirements. If necessary, we can always express an aberrant partial order as the union of several valid partial orders, as we did with the tree example in equation (2.9).

2.4.1 Left to Right Maxima and Minima

Problem. *How many permutations of 1 to n have j left to right maxima and k left to right minima?*

Permutations are encoded in the subscripts of the x's, with an l before each maximum and s before each minimum. For example, there are six strings of length $n = 3$ with $j = 2$ maxima and $k = 2$ minima,

$$\begin{array}{ll} slx_3sx_1x_2lx_4 & slx_2lx_4x_3sx_1 \\ slx_3sx_1lx_4x_2 & slx_2lx_4sx_1x_3 \\ slx_3lx_4sx_1x_2 & slx_2sx_1lx_4x_3 \end{array} \tag{2.89}$$

The grammar,

$$\begin{aligned} T &\to \{e > f; e > g\}\,[b < a; b < c]\, T_{[a]}^{\{e\}} sx_{[b]}^{\{f\}} R_{[c]}^{\{g\}} \\ T &\to \{f > e; f > g\}\,[a < b; a < c]\, T_{[a]}^{\{e\}} lx_{[b]}^{\{f\}} R_{[c]}^{\{g\}} \\ T &\to slx \\ R &\to xR \,|\, \epsilon, \end{aligned} \tag{2.90}$$

is probably clearer if we adopt the convention that $\blacksquare$ marks the location of the largest label:

$$\begin{aligned} T &\to T^{\blacksquare} sx^{\square} R \\ T &\to T^{\square} lx^{\blacksquare} R \\ T &\to slx \\ R &\to xR \,|\, \epsilon. \end{aligned} \tag{2.91}$$

Permutations are decomposed based on the location of their smallest and largest elements. The first production for T covers situations where the smallest element is to the right of the largest element; the smallest will be a minimum, so it is prefixed with an s. The second T production handles those cases where the largest is right of the smallest. R generates an arbitrary string of x's, for it is to the right of both the smallest and largest elements and so can have no further maxima or minima.

The partial orders are well separated, so, using the translation defined above, we discover

$$\begin{aligned} T &= \iint (T' sR) + \iint (T' lR) + slx \\ R &= xR + 1 \\ R &= \frac{1}{1-x} \\ T &= \iint T' \left(\frac{s+l}{1-x}\right) + slx \\ T'' &= \left(\frac{s+l}{1-x}\right) T' \\ \ln T' &= -(s+l)\ln(1-x) + c_1 \\ T' &= c_2 (1-x)^{-(s+l)}. \end{aligned} \tag{2.92}$$

We conclude that $c_2 = sl$ since the only string with a single x has both a maximum and a minimum at that x. The grammar cannot derive an empty string, so there is no constant term in the final generating function:

$$T = \frac{sl}{s+l-1}\left((1-x)^{-s-l+1} - 1\right). \tag{2.93}$$

To derive strings representing permutations we need partial orders that are all 1-smallest and 1-largest determined. This is not true of the last production of (2.90), but the failure is easily remedied; we expand the production into several possibilities:

$$\begin{aligned} R &\to x_{[h]}^{\{j\}} \\ R &\to \{k > j\}\,[h < i]\, x_{[h]}^{\{j\}} R_{[i]}^{\{k\}} \\ R &\to \{j > k\}\,[i < h]\, x_{[h]}^{\{j\}} R_{[i]}^{\{k\}} \\ R &\to \{k > j\}\,[i < h]\, x_{[h]}^{\{j\}} R_{[i]}^{\{k\}}. \end{aligned} \tag{2.94}$$

Now every production is 1-smallest and 1-largest determined. We can work a short example by starting with the first production of (2.90),

$$\{e > f; e > g\}\,[b < a; b < c]\, T_{[a]}^{\{e\}} s x_{[b]}^{\{f\}} R_{[c]}^{\{g\}}, \tag{2.95}$$

deriving a slx from the T nonterminal,

$$\{e > f; e > g\}\,[b < a; b < c]\, slx_{[a]}^{\{e\}} s x_{[b]}^{\{f\}} R_{[c]}^{\{g\}}, \tag{2.96}$$

and then expanding R with the production $R \to x_{[h]}^{\{j\}}$:

$$\{e > f; e > g\}\,[b < a; b < c]\, slx_{[a]}^{\{e\}} s x_{[b]}^{\{f\}} x_{[c]}^{\{g\}}. \tag{2.97}$$

There is only one way to label the remaining string that is consistent with both partial orders:

$$slx_3 s x_1 x_2. \tag{2.98}$$

The fact that there seems to be one possibility,

$$R \to \{j > k\}\,[h < i]\, x_{[h]}^{\{j\}} R_{[i]}^{\{k\}}, \tag{2.99}$$

missing from the productions for R is extremely illustrative. First, notice that this possibility is not a valid production. The special character x has two labels, neither of which are boundary elements of their respective partial orders. Second, notice that of all the possibilities for R at (2.94), only the invalid production given here at (2.99) can be followed by the replacement $R \to \epsilon$, since i and k are both boundary elements. In fact, if we apply $R \to \epsilon$ to (2.99) we obtain the correct production $R \to x_{[h]}^{\{j\}}$ found in (2.94). So the rewriting of R to meet the 1-determinedness requirements is hardly very mysterious after all.

CHAPTER 3

ANALYSIS OF ALGORITHMS

In the past decade there has been rapid growth in a subspecialty of theoretical computer science called *analysis of algorithms*. Research has focused on the careful computation of average and worse case running times for algorithms. This is to be distinguished from *mathematical theory of computation*, which is more concerned with the logic and correctness of algorithms, and *computational complexity*, where the analysis is aimed at broader classifications like separating $O(n^2)$ algorithms from $O(n \ln n)$. By contrast, *analysis of algorithms* endeavors to find exact or detailed asymptotic expansions for the performance of algorithms. On the theoretical side, this attention to detail leads to a great variety of interesting combinatorial mathematics, while on the practical side it can determine such questions as when an $O(n \ln n)$ algorithm surpasses an $O(n^2)$ algorithm. In this way the analysis of algorithms spans both the theoretical and practical worlds of computer science.

Figure 3.1 below is a rough diagram of the process of analyzing an algorithm. There are two paths. Across the bottom is what might be called the direct approach: within the algorithm a quantity of interest is identified, a recurrence relation is derived, and the recurrence is solved, using manipulations of discrete mathematics. The other, seemingly circuitous path through analytic equations and generating functions is often the easiest method of solution, because once recurrence relations are converted to functional equations the solution strategy is usually routine; we are solving quadratic or differential equations rather than manipulating sums of binomial coefficients, and so the techniques of real analysis can be effectively applied to discrete problems.

Labelled formal languages are useful for shifting from the discrete domain to the continuous domain of real analysis. We have explored already in Chapter 2 the conversion of grammars to differential equations, this is the transition in the upper left hand corner of Figure 3.1 and undoubtedly the simplest step in the whole diagram. However, the analyst needs to express some quantity of interest in the algorithm using a formal language. This requires cleverness—it is not an automated step—but since the language of expression is based on formal languages it is a natural means of expression for computer scientists.

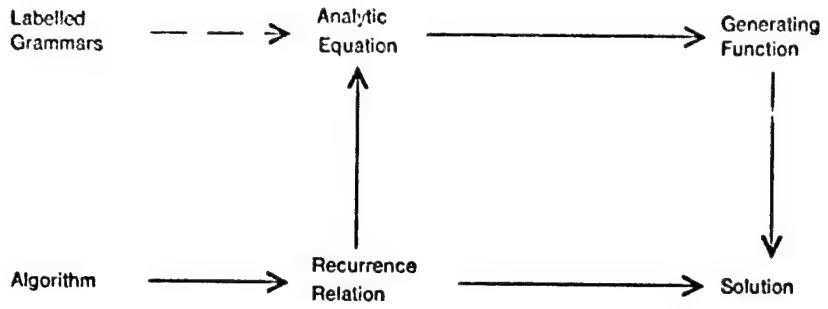


Figure 3.1 A Simplified Picture of the Analysis of an Algorithm

The purpose of this chapter is to demonstrate the usefulness of labelled formal languages in the analysis of algorithms. To some extent this has been demonstrated already in Chapter 2, since variations of the classic generating functions appear frequently in the analysis of algorithms. To press the point further, the examples chosen for this chapter use labelled formal languages in a central point of their analysis, and use them in a way that is more complex than the examples of Chapter 2. We will study cache memories and search trees.

3.1 The Degree of Associativity of a Hardwired Cache Memory

Because a computer program is likely to make frequent reference to a small subset of its total address space, computer designers have discovered that substantial cost savings are possible with memory hierarchies. The idea is to put a frequently accessed subset of the address space into a small, fast, and expensive memory. Most memory references will reside in this *cache* memory, so the performance of the computer will be similar to a computer whose whole memory is built in this fast expensive manner, while the cost of the computer will in fact be based on a slower, cheaper main memory.

Fitting the whole address space into a small cache is difficult: one approach is to use a fully associative cache that stores both the address and the content of each main memory entry residing in the cache. The cache is capable of simultaneously comparing a requested address with the addresses of each of the entries in the cache. If the requested address in present in the cache its contents are made available for processing, otherwise, a slower process fetches the entry from main memory.

A second approach to caching saves most of the hardware necessary for parallel comparisons. Part of the memory address is used as an index into a table. Within a particular slot of this table several memory entries are stored along with the remainder of their memory addresses. To search this table for a requested address we find the appropriate slot, read the entire slot into a special area and search the slot associatively for the desired entry. This is pictured in Figure 3.2.

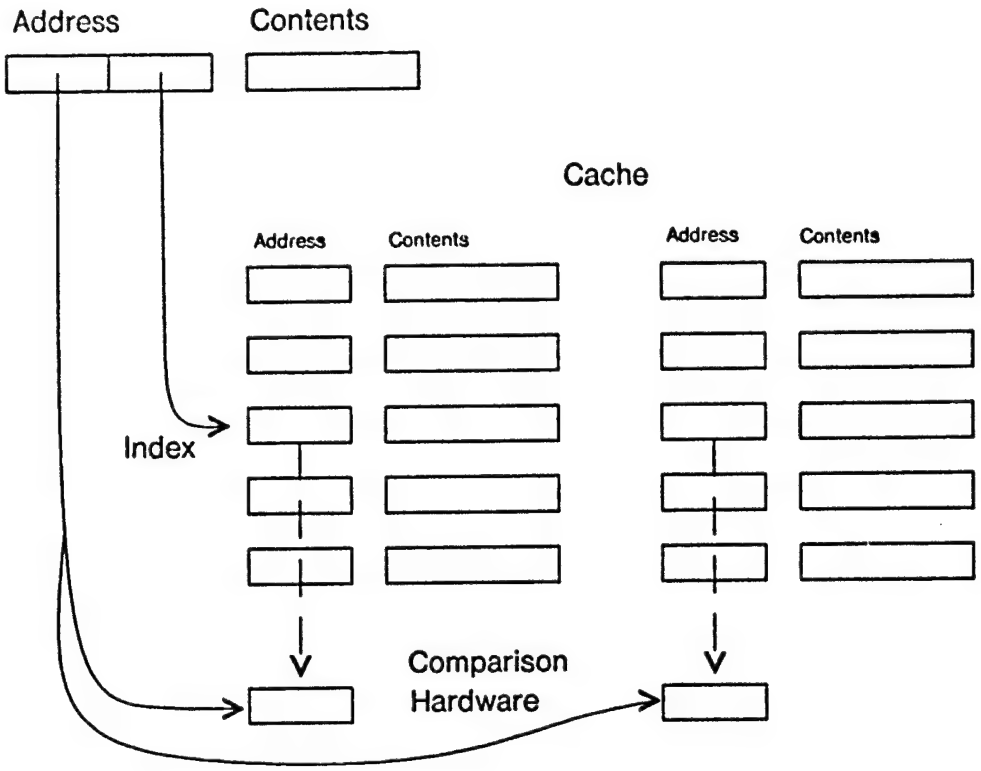


Figure 3.2 A Two Way Associative Cache Memory

The number of entries in each slot of the cache is the degree of associativity. It is a design parameter affecting both the performance of the cache, that is, how close it comes to a perfectly associative scheme, and the cost of the cache since it determines the amount of comparison hardware needed. Empirical studies indicate that very little is gained by increasing the associativity beyond 2 or 4; however, we would like to understand the functional relationship between associativity and performance. To do this we will use the following simplified model.

Assume that a load of size L is chosen at random from the total address space. We compute the performance of the load by inserting the load items into the cache. Some slots will overflow, in which case the extra entries remain in main memory. We assume that the program references the load items uniformly at random, so the performance depends on the amount of cache overflow. (This assumption makes the analysis independent of the replacement algorithm, since "uniformly at random" ignores the gain of moving entries into the cache after they are referenced. The locality of program reference is modelled instead by a choice of small L.)

Under these assumptions we obtain an occupancy problem. If I is the index size and A the degree of associativity then the corresponding occupancy problem involves loading

L numbered objects into I boxes, tagging each extra object after the Ath with an overflow indicator.

Here is a grammar describing this process for $A = 2$:

$$\begin{aligned} S &\to B_2 p S \mid \epsilon \\ B_2 &\to x^{\square} B_1 \mid \epsilon \\ B_1 &\to x^{\square} B_0 \mid \epsilon \\ B_0 &\to o x^{\square} B_0 \mid \epsilon . \end{aligned} \tag{3.1}$$

The p's separate boxes, the x's are the objects, and a prefix o indicates that an object has overflowed. The grammar's equations can be solved:

$$\begin{aligned} B_0 &= e^{ox} \\ B_1 &= \frac{e^{ox} - 1}{o} + 1 \\ B_2 &= \frac{e^{ox}}{o^2} - \frac{x}{o} - \frac{1}{o^2} + x + 1 \\ S &= \frac{1}{1 - B_2 p} . \end{aligned} \tag{3.2}$$

From this construction it is clear how the associativity affects the generating function:

1) One way.

$$B_1 = \frac{e^{ox}}{o} - \frac{1}{o} + 1 \tag{3.3}$$

2) Two way.

$$B_2 = \frac{e^{ox}}{o^2} - \frac{x}{o} - \frac{1}{o^2} + x + 1 \tag{3.4}$$

3) A way.

$$B_A = \frac{e^{ox}}{o^A} - \frac{x^{A-1}}{o(A-1)!} - \frac{x^{A-2}}{o^2(A-2)!} - \cdots - \frac{1}{o^A} + \frac{x^{A-1}}{(A-1)!} + \frac{x^{A-2}}{(A-2)!} + \cdots + 1 \tag{3.5}$$

But it is not clear that we can obtain any meaningful information from these formulas. S is a multivariate function of p, o, and x. The coefficient of $p^I \frac{x^L}{L!}$ in S is a generating function in o representing the distribution of overflow for load L in I boxes. We can gain information about overflow by differentiating with respect to o first before we extract the coefficient of $p^I \frac{x^L}{L!}$:

$$\text{Average overflow} = \frac{1}{I^L} \left\langle p^I \frac{x^L}{L!} \right\rangle \frac{\partial}{\partial o} S(p, x, o) \Big|_{o=1} . \tag{3.6}$$

The equation is divided by all possible loadings of objects into boxes (I^L) in order to obtain a probability distribution. This is also accomplished with the substitution $x \leftarrow y/I$:

$$\begin{aligned}\text{Average overflow} &= \left\langle p^I \frac{y^L}{L!} \right\rangle \frac{\partial}{\partial o} S(p, y/I, o)\Big|_{o=1} \\ &= \left\langle \frac{y^L}{L!} \right\rangle \frac{\partial}{\partial o} B_A(y/I, o)^I\Big|_{o=1} \\ &= \left\langle \frac{y^L}{L!} \right\rangle I B_A(y/I, 1)^{I-1} \frac{\partial}{\partial o} B_A(y/I, o)\Big|_{o=1}.\end{aligned} \tag{3.7}$$

When $o = 1$, the expression for $B_A(y/I, o)$ collapses nicely:

$$B_A(y/I, 1) = e^{y/I}. \tag{3.8}$$

The complexity lies in

$$\frac{\partial}{\partial o} B_A(y/I, o) = \frac{o^A e^{oy/I} y/I - A o^{A-1} e^{oy/I}}{o^{2A}} + \frac{(y/I)^{A-1}}{o^2(A-1)!} + 2\frac{(y/I)^{A-2}}{o^3(A-2)!} + \cdots + A\frac{1}{o^{A+1}}. \tag{3.9}$$

Putting this together,

$$\text{Average overflow} = \left\langle \frac{y^L}{L!} \right\rangle I e^{y(I-1)/I} \left(e^{y/I}\frac{y}{I} - A e^{y/I} + \frac{(y/I)^{A-1}}{(A-1)!} + 2\frac{(y/I)^{A-2}}{(A-2)!} + \cdots + A \right), \tag{3.10}$$

thus for $A = 2$ we expect the overflow to be

$$L - 2I + L(1 - 1/I)^{L-1} + 2I(1 - 1/I)^L. \tag{3.11}$$

A similar calculation, using the second derivative with respect to o, yields the variance.

For comparison of different A a realistic standard is the total volume $V = AI$. We assume that the load is some proportion of this volume $L = \alpha V$. As $\alpha \to \infty$, $L - V$ overflow is expected, regardless of the associativity. For smaller α the associativity plays a more important rôle. To examine this region we let $V \to \infty$ and approximate $(1 - 1/I)^L$ with $e^{-\alpha A}$:

$$\begin{aligned}\text{Average overflow} &= L - V + I e^{-\alpha A}\left(\frac{1}{(A-1)!}\left(\frac{L}{I}\right)^{A-1} + \frac{2}{(A-2)!}\left(\frac{L}{I}\right)^{A-2} + \cdots \right. \\ &\qquad \left. + \frac{A-1}{1!}\left(\frac{L}{I}\right) + A \right) + O\left(\frac{1}{V}\right) \\ &= L - V + I e^{-\alpha A}\left(\frac{(\alpha A)^{A-1}}{(A-1)!} + \frac{2(\alpha A)^{A-2}}{(A-2)!} + \cdots + A \right) + O\left(\frac{1}{V}\right).\end{aligned} \tag{3.12}$$

Two cases are of interest. When $\alpha \to 0$, the overflow is exponentially related to the associativity:

$$\begin{aligned}\text{Average overflow} &= V\left(\alpha - 1 + \frac{e^{-\alpha A}}{A}\left(Ae^{\alpha A} - \alpha A e^{\alpha A} + \frac{(\alpha A)^{A+1}}{(A+1)!} + \frac{2(\alpha A)^{A+2}}{(A+2)!} + \cdots\right)\right) \\ &= V\left(\frac{e^{-\alpha A}(\alpha A)^{A+1}}{A(A+1)!}\right) E(\alpha),\end{aligned} \tag{3.13}$$

where the error is constrained by

$$1 \le E(\alpha) \le \frac{1}{(1-\alpha)^2}. \tag{3.14}$$

For small α the most sensitive term in equation (3.13) is α^{A+1}; so the percentage overflow falls off dramatically with the associativity.

A second case of interest is when $\alpha = 1$, corresponding to a cache pressed nearly to its limit:

$$\begin{aligned}\text{Average overflow} &= V\left(\alpha - 1 + e^{-\alpha A}\left(\frac{(\alpha A)^{A-1}}{(A-1)!} + \frac{(\alpha A)^{A-2}}{(A-2)!} + \cdots + 1\right)\right. \\ &\qquad \left. - \frac{e^{-\alpha A}}{A}\left(\frac{(A-1)(\alpha A)^{A-1}}{(A-1)!} + \frac{(A-2)(\alpha A)^{A-2}}{(A-2)!} + \cdots + 0\right)\right) \\ &= V\left(\alpha - 1 + (1-\alpha)e^{-\alpha A}\left(\frac{(\alpha A)^{A-2}}{(A-2)!} + \frac{(\alpha A)^{A-3}}{(A-3)!} + \cdots + 1\right)\right. \\ &\qquad \left. + e^{-\alpha A}\frac{(\alpha A)^{A-1}}{(A-1)!}\right) \\ &= V e^{-A}\frac{A^{A-1}}{(A-1)!} \\ &= V\frac{1}{\sqrt{2\pi A}}\left(1 + O\left(\frac{1}{A}\right)\right).\end{aligned} \tag{3.15}$$

Here the percentage overflow falls off inversely with the square root of the associativity.

Depending on one's temperament, there are two ways of looking at this last result. Remember that the case $\alpha = 1$ causes a fully associative cache to work beautifully, with no overflow and full memory usage. The above result gives encouraging news: a cheaper, four way associative cache is worse by only 20% overflow. However, the result also gives discouraging news: when the load factor α nears 1 increasing the degree of associativity is not particularly helpful. It is only when α is small and performance good anyway that the degree of associativity has an impressive effect on the amount of overflow.

Let us pause for a moment to compare the augmented grammar approach with the traditional analysis of this problem. Normally we would assume that the number of boxes is large so that we could approximate the behavior of an individual box with the Poisson distribution. (In fact, fragments of the Poisson distribution appear in the above formulas.) Then we would sum the expected overflow from each box to obtain an average overflow

(a technique that wouldn't work for the variance). The chief advantage of the augmented grammar approach is that it leads gracefully to a multivariate generating function that contains all the important information, without approximations. Approximations can come later in the process of understanding the asymptotic features of the distribution.

3.2 Binary Tree Search

A binary tree is a data structure used for the storage and retrieval of information based on keys associated with entries in the tree. The data structure is well suited for applications where:

1) There are roughly the same number of insertions as retrievals.

2) It is difficult to predict how large the data structure will grow.

3) Average performance is more important than worse case behavior.

4) The data structure resides in main memory.

5) It is helpful to know the order of keys.

If any of these criteria are not met then there are better options: when retrievals greatly exceed insertions then a balanced tree scheme is preferable. Hashing is used in cases where the table size is fixed in advance, and the order of keys is irrelevant. When the data size exceeds main memory it is often better to use a structure adapted to the computer's page size, such as B-trees. Nevertheless there is a distinct domain where binary trees are the best known method of storage.

This section proposes a modification that is intermediate between balanced and unbalanced tree searching. Rather than completely balancing the tree, it only improves the balance of the data structure; we will call it diminished tree searching. The domain of usefulness for diminished tree search overlaps primarily with the ordinary tree search domain described above; and, as we will see shortly in the analysis, it outperforms ordinary tree search on the larger problems.

3.2.1 Diminished Trees

In ordinary binary tree search the tree nodes contain a left link, a right link and a key; and the central loop of the search routine looks something like this:

```
while true do
  if t↑.key > sought then
    if t↑.left ≠ nil then t ← t↑.left else goto missing
  else if t↑.key < sought then
    if t↑.right ≠ nil then t ← t↑.right else goto missing
  else goto found
missing: ...
found: ...
```

Notice that there are two exits from the loop, one for a successful search, and one for the unsuccessful possibility that the search encounters a nil pointer. Since the loop will usually be executed a logarithmic number of times it represents the largest asymptotic contribution to the running time of the algorithm. The exits require only a constant amount of additional computation and are therefore less important to the overall performance of the algorithm.

Recognizing this difference in cost, the modifications necessary for diminishing the search path are made entirely within the exit code, leaving the inner loop of the search in its swift untouched form. To do this we introduce dummy nodes at the leaves of the tree:

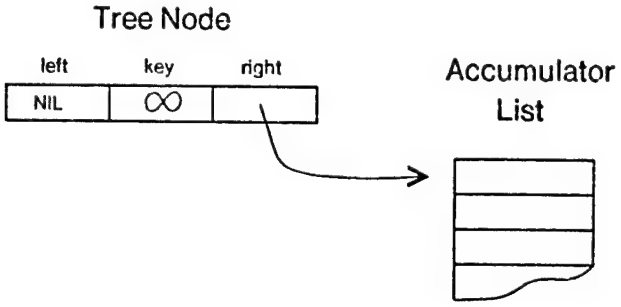


Figure 3.3 A Dummy Node and an Accumulator

Due to the infinitely large key at a dummy node, the search loop will always turn left, encounter a nil pointer, and take the "unsuccessful" loop exit, at which point new code will pick up the right pointer and search the accumulator list for the desired key. The accumulator is maintained as an ordered list, so if the sought key is missing and must be inserted then shifting may be necessary to keep the keys ordered. The accumulator also has a fixed odd size, $2t+1$. If the arriving key fills the accumulator then the list is split, the median key is placed in a new tree node, and two accumulators are created with t elements each. This is depicted in Figure 3.4. The algorithmic details can be found in Appendix B.

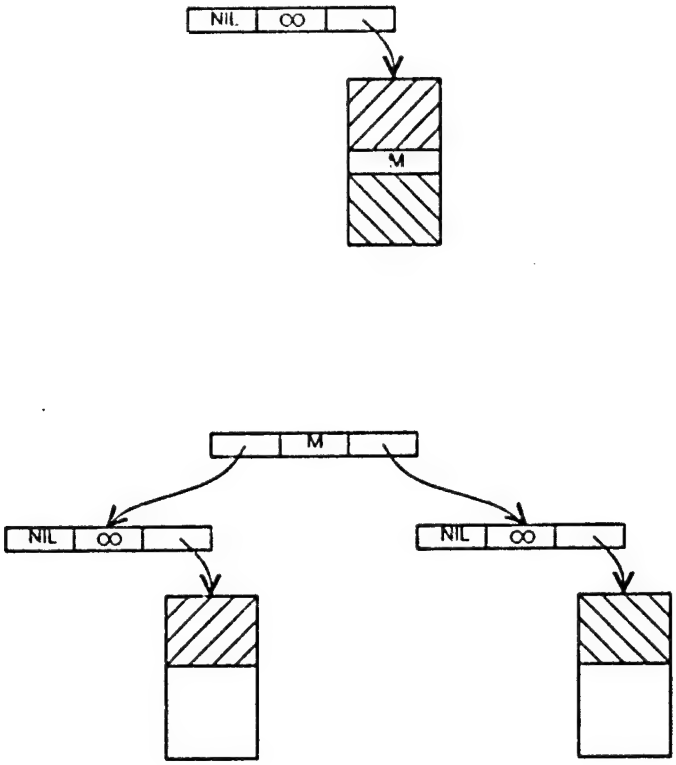


Figure 3.4 An Accumulator Splits

The accumulators allow the tree to grow in a delayed fashion. Those keys appearing in tree nodes are chosen after $2t + 1$ keys are examined. This extra enlightenment means that tree nodes will more evenly split future insertions into their subtrees and so diminish the path length of the whole tree. However, we need to study this change carefully to understand when path length improvements exceed additional accumulator costs and to pick the best value of the parameter t.

3.2.2 Analysis of Diminished Tree Searching

For the analysis it is convenient to introduce the notion of an ordered heap, an object that several authors have found useful in studying binary trees [Burge 1972] [Françon 1976] [Viennot 1976]. A heap is a binary tree with the labels 1 to n assigned to the nodes in such a way that each node has a smaller label than all of its descendents. An example is pictured in Figure 3.5.

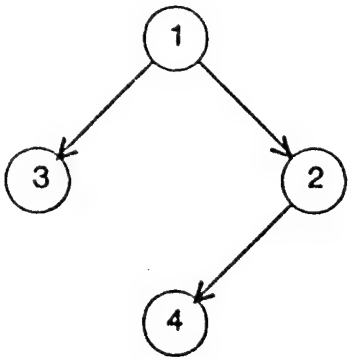
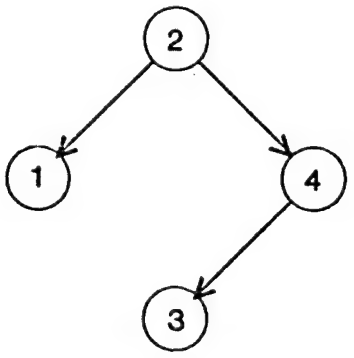


Figure 3.5 A Heap

Search trees and heaps are related as follows. We take a permutation of 1 to n and insert it into an ordinary binary search tree. At the same time we create a heap with an identical shape as the search tree, but we use the order of filling the search tree to label the nodes of the heap. Figure 3.6 shows the permutation and search tree corresponding to the heap in Figure 3.5.



2413

Figure 3.6 The Companion Search Tree and Permutation for Figure 3.5

Given a heap we can recover the associated permutation by walking the heap in infix order. If the ith node in infix order has label j then i is inserted into position j of the permutation. There is thus a one to one correspondence between heaps and permutations that will make heaps especially useful in analysis.

Let us begin our analysis by using heaps to study ordinary binary trees. The grammar for a heap uses the box operator to insure that every node's label is smaller than those of its descendants:

$$S \rightarrow x^{\square} lShlSh \mid \epsilon. \tag{3.16}$$

And, not surprisingly, the solution of this grammar indicates that there are $n!$ heaps of size n:

$$\begin{aligned} S &= \int S^2 + 1 \\ &= \frac{1}{1-x}. \end{aligned} \tag{3.17}$$

Of greater interest is the path length of a heap. For any rooted tree the (internal) path length is defined to be the sum of the distances from each node to the root. In the case of heaps the path length is the total number of comparisons used to build the corresponding search tree. Dividing by n, the size of the tree, and adding one for the last comparison, gives the average number of comparisons for a successful search.

To analyze path length we add an index i for depth:

$$S_i \to q^i x \square l S_{i+1} h l S_{i+1} h \mid \epsilon. \tag{3.18}$$

Each node x has been prefaced by a string of q's recording its depth in the tree.

Converting this to an integral equation,

$$S_i = q^i \int S_{i+1}^2 + 1, \tag{3.19}$$

and applying the Q operator ($Qf(x) = f(qx)$) yields the the same equation as for S_{i+1}:

$$\begin{aligned} QS_i &= q^i Q \int S_{i+1}^2 + 1 \\ &= q^{i+1} \int (QS_{i+1})^2 + 1. \end{aligned} \tag{3.20}$$

This suggests the conjecture $QS_i = S_{i+1}$, which agrees with our intuition: any subtree with root at depth i can be moved to depth $i+1$ by prefacing each node x with an extra q.

For the whole tree we obtain the equation

$$S_0' = (QS_0)^2 \tag{3.21}$$

which does not seem to have a simple solution. With the chain rule of differentiation, however, we can extract useful information from the equation. To obtain the mean path length, we differentiate with respect to q, seeking a generating function in x alone:

$$T(x) = \frac{\partial}{\partial q} S_0(x,q)\Big|_{q=1}. \tag{3.22}$$

The coefficient of x^i in $T(x)$, treated as an ordinary generating function in x, will be the average path length of heaps of size i. (Ordinarily all generating functions in the labelled terminal x are treated as exponential. In this situation the $i!$ is the normalizing factor for the probability distribution in path length.)

Differentiating both sides of (3.21) with respect to q and using the chain rule reveals an ordinary differential equation for T,

$$\frac{\partial}{\partial q}\frac{\partial}{\partial x}S_0(x,q)\Big|_{q=1} = 2\,S_0(qx,q)\left[x\frac{\partial}{\partial x}S_0(x,q) + \frac{\partial}{\partial q}S_0(x,q)\right]\Big|_{q=1} \tag{3.23}$$

$$T' = \frac{2}{1-x}\left[x\frac{1}{(1-x)^2} + T(x)\right] \tag{3.24}$$

$$(1-x)T' - 2T(x) = \frac{2x}{(1-x)^2}. \tag{3.25}$$

With integrating factor $(1-x)^{-2}$,

$$\begin{aligned} T(x) &= (1-x)^{-2}G(x) \\ G'(x) &= \frac{2x}{1-x} \\ G(x) &= -2\ln(1-x) - 2x \\ T(x) &= -2\frac{\ln(1-x)+x}{(1-x)^2}, \end{aligned} \tag{3.26}$$

and we can recover the coefficient of x^i using a family of identities, due to Zave [Zave 1976], for powers of $\ln(1-x)$ over powers of $(1-x)$. In this case,

$$\frac{-\ln(1-x)}{(1-x)^2} = \sum_{n\geq 0}(H_{n+1}-1)(n+1)z^n, \tag{3.27}$$

so the expected path length is

$$2(H_{n+1}-1)(n-1) - 2n, \tag{3.28}$$

and the average successful search will require

$$2H_n - 3 + 2\frac{H_n}{n} \tag{3.29}$$

comparisons.

The intent of this section was to study diminished searching and yet so far we have only managed to compute the comparisons for ordinary tree search. However, the steps of the above analysis will provide a model for our analysis of diminished searching. Recall that we:

1) Established a correspondence between the data structure of interest (search trees) and another structure (heaps) that logged the arrival of elements of the first structure.

2) Developed a labelled grammar and converted it to a differential equation in two variables.

3) Removed one variable (using the Q operator, and the chain rule), and solved the remaining differential equation for a generating function.

4) Recovered the coefficients of the generating function.

These steps will be repeated in the analysis of Diminished Tree Search, but first let us review some mathematics that will prove useful in the analysis that follows.

In the fourth step we made use of an identity of Zave. Here is the most general form of the identity:

$$\frac{(-\ln(1-z))^n}{(1-z)^{m+1}} = \sum_{j\geq 0} P_n(H_{m+j} - H_m, \ldots, H^{\langle n\rangle}_{m+j} - H^{\langle n\rangle}_m)\binom{m+j}{m} z^k, \tag{3.30}$$

where the angle bracketed superscripts indicate a truncated Riemann sums,

$$H^{\langle k\rangle}_n = 1 + \frac{1}{2^k} + \frac{1}{3^k} + \cdots + \frac{1}{n^k}, \tag{3.31}$$

and the polynomials P_n are related to the Bell polynomials Y_n:

$$P_n(s_1, \ldots, s_n) = (-1)^n Y_n(-s_1, -s_2, -2s_3, \ldots, -(n-1)!\, s_n). \tag{3.32}$$

We will make heavy use of two subcases:

$$\frac{-\ln(1-z)}{(1-z)^{m+1}} = \sum_{j\geq 0}(H_{m+j} - H_m)\binom{m+j}{m} z^j \tag{3.33}$$

$$\frac{(\ln(1-z))^2}{(1-z)^{m+1}} = \sum_{j\geq 0}((H_{m+j} - H_m)^2 - (H^{\langle 2\rangle}_{m+j} - H^{\langle 2\rangle}_m))\binom{m+j}{m} z^j, \tag{3.34}$$

in order to recover the coefficients of generating functions.

In the third step we solved a differential equation,

$$(1-x)T' - 2T = \frac{2x}{(1-x)^2}, \tag{3.35}$$

by using the integrating factor $(1-x)^{-2}$. Such a clean integrating factor was possible because of the $(1-x)$ factor prefixing T'. In general, we can solve higher order equations like

$$a(1-x)^3T''' + b(1-x)^2T'' + c(1-x)T' + dT = 0 \tag{3.36}$$

as long as the power of the prefixing factor matches the differentiation of T. First we use the change of variables $v = 1-x$, and then we replace differentiation with the operator $\vartheta = v\frac{\partial}{\partial v}$, so a term like $v^nT^{(n)}$ can be expressed with a falling factorial of the operator: $\vartheta^{\underline{n}}T^{(n)}$. Equation (3.36) above becomes

$$(-a\,\vartheta^{\underline{3}} + b\,\vartheta^{\underline{2}} - c\,\vartheta + d)T = 0. \tag{3.37}$$

In general we will have a polynomial of the operator applied to T:

$$P(\vartheta)T = 0\,. \tag{3.38}$$

Suppose we have a factorization of this polynomial and r is one of roots. The contribution of a single root can be found using v^r for an integrating factor:

$$\begin{aligned}(\vartheta - r)G &= 0\\ G &= v^r H\\ v^r \vartheta H &= 0\\ \frac{\partial H}{\partial v} &= 0\\ H &= k\\ G &= kv^r\,,\end{aligned} \tag{3.39}$$

where k is a constant determined from initial conditions. If the equation is inhomogeneous, with a right hand side of v^s, $s \neq r$, then the solution is still straightforward:

$$\begin{aligned}(\vartheta - r)G &= v^s\\ G &= v^r H\\ v^r \vartheta H &= v^s\\ \frac{\partial H}{\partial v} &= v^{s-r-1}\\ H &= k + \frac{v^{s-r}}{(s-r)}\\ G &= kv^r + \frac{v^s}{(s-r)}\,.\end{aligned} \tag{3.40}$$

By combining these two solutions, we can see the effect of a simple root in a higher degree polynomial. Let the polynomial have distinct roots:

$$P(\vartheta) = (\vartheta - r_1)(\vartheta - r_2)\ldots(\vartheta - r_n)\,. \tag{3.41}$$

Then $P(\vartheta)T = 0$ can be solved by setting $G = (\vartheta - r_2)(\vartheta - r_3)\ldots(\vartheta - r_n)T$. The equation becomes

$$(\vartheta - r_1)G = 0 \tag{3.42}$$

with solution $G = kv^{r_1}$. Now the remainder of the equation is inhomogeneous,

$$(\vartheta - r_2)(\vartheta - r_3)\ldots(\vartheta - r_n)T = kv^{r_1}\,, \tag{3.43}$$

but we can continue to strip factors from the beginning of the equation until we eventually obtain

$$T = \frac{k}{R(r_1)}\,v^{r_1} + \cdots\,, \tag{3.44}$$

where $R(\vartheta) = P(\vartheta)/(\vartheta - r_1)$, and the other roots contribute terms that are not shown in equation (3.44).

The solution of an inhomogeneous equation with a right hand side of v^r is slightly more complicated:

$$\begin{aligned}(\vartheta - r)G &= v^r \\ G &= v^r H \\ v^r \vartheta H &= v^r \\ \frac{\partial H}{\partial v} &= v^{-1} \\ H &= \ln v + k \\ G &= v^r \ln v + k v^r .\end{aligned} \tag{3.45}$$

So we will also need to be able to deal with logarithms on the right hand side:

$$\begin{aligned}(\vartheta - r)G &= v^s \ln v \\ G &= v^r H \\ v^r \vartheta H &= v^s \ln v \\ \frac{\partial H}{\partial v} &= v^{s-r-1} \ln v \\ H &= \frac{v^{s-r} \ln v}{(s-r)} - \frac{v^{s-r}}{(s-r)^2} + k \\ G &= \frac{v^s \ln v}{(s-r)} - \frac{v^s}{(s-r)^2} + k v^r .\end{aligned} \tag{3.46}$$

Once again we can combine these two solutions to see the effect of a single root that is equal to the power of the inhomogeneous right hand side. We assume that the other roots are distinct:

$$\begin{aligned}P(\vartheta) &= (\vartheta - r_1)(\vartheta - r_2)\dots(\vartheta - r_n) \\ P(\vartheta)T &= v^{r_1} \\ G &= (\vartheta - r_2)(\vartheta - r_3)\dots(\vartheta - r_n)T \\ (\vartheta - r_1)G &= v^{r_1} \\ G &= v^{r_1} \ln v + k v^{r_1} .\end{aligned} \tag{3.47}$$

Repeating this technique for the other roots modifies the constants on the contribution of the first root until finally,

$$T = \frac{v^{r_1} \ln v}{R(r_1)} - \frac{R'(r_1)}{(R(r_1))^2} v^{r_1} + \frac{k}{R(r_1)} v^{r_1} + \cdots , \tag{3.48}$$

where again, $R(\vartheta) = P(\vartheta)/(\vartheta - r_1)$ and k is a constant determined by initial conditions.

With the mathematical preliminaries completed we can begin the analysis of Diminished Tree Search. The data structure is encoded with the usual l and h characters for the tree portion of the structure. The x characters are used for valid keys present in the internal tree nodes, and y's are used for the dummy $+\infty$ keys in the leaves. The accumulator lists appear as contiguous blocks of x's, enclosed with b's and e's. Ignoring the labelling of the x's for the moment, the following grammar encodes the shape of a diminished tree:

$$\begin{aligned} S &\to xlShlSh \mid ylhlAh \\ A &\to bxxe \mid bxxxe \mid bxxxxe. \end{aligned} \tag{3.49}$$

In this case the accumulator parameter is $t = 2$, so the accumulator lists will range in size from 2 to 4. We assume that the tree always has at least t keys.

Following the analysis completed already for ordinary tree search, the x's are labelled with the order they were first inserted, rather than using the actual keys. The labels are constrained by the construction process for the tree. Since a tree node results from the splitting of an accumulator list of size $2t + 1$, we expect that the smallest $2t + 1$ labels of the subtree (representing the first keys inserted) will be divided evenly: t will appear in the right subtree, t will appear in the left subtree, and one will appear as the key of the tree node. The following partial order expresses the constraint on labels of a subtree:

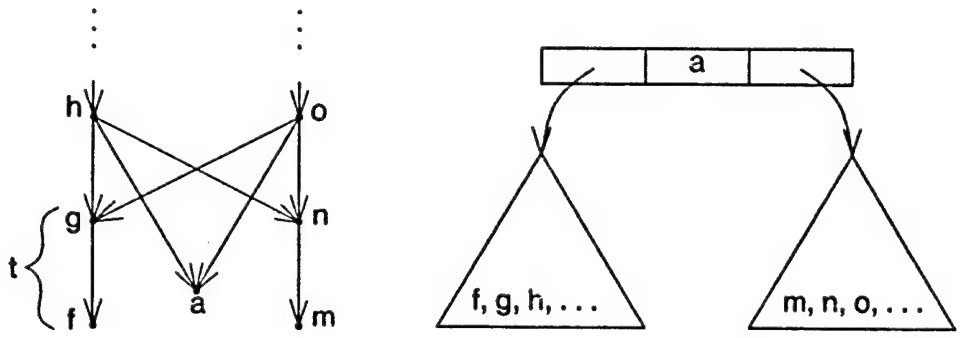


Figure 3.7 The Distribution of Labels in Subtrees

Denoting the partial order by $\mathcal{B}$ we can augment the grammar so that it specifies a labelled formal language:

$$\begin{aligned} S &\to [\mathcal{B}]x_{[a]}lS_{[fgh]}hlS_{[mno]}h \mid ylhlAh \\ A &\to bxxe \mid bxxxe \mid bxxxxe. \end{aligned} \tag{3.50}$$

The partial order $\mathcal{B}$ is well separated: in Figure 3.7 g, f, a, m, and n are active, while h and o are boundary elements. To convert the grammar to an equation we multiply the number of linear embeddings of $\mathcal{B}$, $\binom{2t+1}{t}(t+1)$, by a repeated integral:

$$S = \binom{2t+1}{t}(t+1)\int^{(2t+1)} (S^{(t)})^2 + x^t + x^{t+1} + \cdots + x^{2t}. \tag{3.51}$$

For the moment we have dropped all of the terminal characters from this equation. A superscript in parentheses indicates repeated differentiation unless it appears on an integral sign, in which case it signifies repeated integration. Differentiating both sides of equation (3.51) yields:

$$S^{(2t+1)} = \binom{2t+1}{t}(t+1)(S^{(t)})^2, \tag{3.52}$$

with solution $S = 1/(1-x)$, indicating, as we would expect, that there are $n!$ trees with n keys, one for each permutation. The initial conditions that were present in equation (3.51) but disappeared with the differentiation force us to modify the solution:

$$S = \frac{x^t}{1-x}. \tag{3.53}$$

The first $t-1$ terms of the series are now absent due to the fact that the grammar cannot derive a diminished tree with less than t keys. In practice, of course, the algorithm starts with an empty tree with one accumulator list that absorbs the first few keys. Once this list contains t or more keys, we can claim that all accumulator lists in the tree contain between t and $2t$ keys, so that the above grammar is valid. We could modify the grammar to account for the initial anomaly, but it is easier to use the unmodified equation with the qualification that $n \geq t$. For $n \leq 2t$ the algorithm behaves like an ordinary insertion sort.

Most quantities of interest in the analysis of diminished tree search are obtained from suitable variations of the grammar proposed above. Suppose, for example, we are interested in the number of nodes in a data structure. Ordinary trees contain one node for each key, but diminished trees have dummy nodes with no key at all, and accumulator lists with several keys. The number of nodes corresponds to the number of allocations during the running of the algorithm—an operation that is often expensive in high level programming languages.

To study the number of allocations an c is added to the grammar at each point that a new node is used:

$$\begin{aligned} S &\rightarrow [B]cx_{[a]}lS_{[fgh]}hlS_{[mno]}h \mid cylhlcAh \\ A &\rightarrow bxxe \mid bxxxe \mid bxxxxe. \end{aligned} \tag{3.54}$$

This converts to the equation

$$S^{(2t+1)} = \binom{2t+1}{t}(t+1)c(S^{(t)})^2 \tag{3.55}$$

with initial values

$$S = c^2x^t + c^2x^{t+1} + c^2x^{t+2} + \cdots + c^2x^{2t} + \cdots \tag{3.56}$$

which does not appear to have a nice, closed-form solution. Nevertheless, we can simplify the problem by letting

$$T(x) = \left.\frac{\partial S(x,c)}{\partial c}\right|_{c=1}. \tag{3.57}$$

The coefficient of x^i in $T(x)$ is the average number of nodes in a tree containing i keys. Differentiating equation (3.55) with respect to c gives an equation for T:

$$T^{(2t+1)} = \binom{2t+1}{t}(t+1)\left((S^{(t)}|_{c=1})^2 + 2\,S^{(t)}|_{c=1}\,T^{(t)}\right) \tag{3.58}$$

$$S^{(t)}|_{c=1} = \frac{t!}{(1-x)^{t+1}} \tag{3.59}$$

$$T^{(2t+1)} = \binom{2t+1}{t}(t+1)\left(\frac{(t!)^2}{(1-x)^{2t+2}} + \frac{2t!}{(1-x)^{t+1}}T^{(t)}\right) \tag{3.60}$$

$$(1-x)^{2t+1}T^{(2t+1)} - 2\frac{(2t+1)!}{t!}(1-x)^t T^{(t)} = \frac{(2t+1)!}{1-x}. \tag{3.61}$$

Here $S^{(t)}|_{c=1}$ is obtained by differentiating the solution $S = 1/(1-x)$ of equation (3.52) above.

The substitutions $v = 1 - x$, $\vartheta = v\frac{\partial}{\partial v}$ transform the last equation to a form we've explored already,

$$P_t(\vartheta)\,T = -\frac{(2t+1)!}{v}. \tag{3.62}$$

The polynomial of the operator ϑ depends on the parameter t:

$$P_t(\vartheta) = \vartheta^{\underline{2t+1}} + (-1)^t\,2(2t+1)^{\underline{t+1}}\vartheta^{\underline{t}}. \tag{3.63}$$

This polynomial appears in the analysis of median of n quicksort (see [Sedgewick 1975]), and will appear again in the study of other aspects of diminished searching. It is worth diverting now to study the properties of $P_t(\vartheta)$.

Since $\vartheta^{\underline{t}}$ can be factored from equation (3.63) we know that $\vartheta = 0, 1, 2, \ldots, t-1$ are all roots of $P_t(\vartheta)$, leaving

$$P_t(\vartheta) = \vartheta^{\underline{t}}\left((\vartheta - t)^{\underline{t+1}} + (-1)^t 2(2t+1)^{\underline{t+1}}\right). \tag{3.64}$$

There are other integer roots. Whenever t is odd $\vartheta = 3t + 2$ is a root:

$$\begin{aligned}(\vartheta - t)^{\underline{t+1}} &= (2t+2)^{\underline{t+1}} \\ &= 2(2t+1)^{\underline{t+1}},\end{aligned} \tag{3.65}$$

and $\vartheta = -2$ is also a root, following the observation:

$$\begin{aligned}(\vartheta - t)^{\underline{t+1}} &= (-2-t)^{\underline{t+1}} \\ &= (-1)^{\underline{t+1}}(2t+2)^{\underline{t+1}} \\ &= (-1)^{\underline{t+1}}2(2t+1)^{\underline{t+1}}.\end{aligned} \tag{3.66}$$

As t grows large, we can express the falling factorials of equation (3.66) with gamma functions and use Stirling's approximation:

$$\frac{\Gamma(-t+\vartheta+1)}{\Gamma(-2t+\vartheta)} = \frac{\Gamma(-t-1)}{\Gamma(-2t-2)}$$

$$\frac{e^{t-\vartheta-1}(-t+\vartheta+1)^{-t+\vartheta+1-1/2}}{e^{2t-\vartheta}(-2t+\vartheta)^{-2t+\vartheta-1/2}} = \frac{e^{t+1}(-t-1)^{-t-1-1/2}}{e^{2t+2}(-2t-2)^{-2t-2-1/2}}\left(1+O(t^{-1})\right)$$

$$e^{-t-1}(-t+\vartheta+1)^{t+1}\left(\frac{-t+\vartheta+1}{-2t+\vartheta}\right)^{-2t+\vartheta-1/2} = e^{-t-1}(-t-1)^{t+1}2^{2t+2+1/2}\left(1+O(t^{-1})\right)$$

$$\left(1-\frac{\vartheta+2}{t+1}\right)^{t+1} 2^{-2t-2-1/2}\left(2+\frac{\vartheta+2}{t-\vartheta-1}\right)^{2t-\vartheta+1/2} = 1+O(t^{-1}) \tag{3.67}$$

$$2^{-\vartheta-2} = 1+O(t^{-1})$$

So the other roots converge towards the points $-2 \pm k\frac{2\pi i}{\ln 2}$ as shown in Figure 3.8, where the smooth curves sketch the discrete trajectories of the roots as they were located with MACSYMA's root finding algorithm.

The root with the smallest real component makes the largest contribution to the asymptotic growth of the coefficients of the solution. In the case of $P_t(\vartheta)$, $\vartheta = -2$ is the dominant root, so it will prove useful later to factor $\vartheta + 2$ from the polynomial:

$$\begin{aligned}\frac{(\vartheta-t)^{\underline{t+1}}}{\vartheta+2} &= (\vartheta-t)^{\underline{t}}\frac{(\vartheta-2t)}{\vartheta+2}\\ &= (\vartheta-t)^{\underline{t}} - (2t+2)(\vartheta-t)^{\underline{t-1}}\frac{(\vartheta-2t-1)}{\vartheta+2}\\ &\vdots\\ &= \sum_{j=0}^{t}(-1)^j(2t+2)^{\underline{j}}(\vartheta-t)^{\underline{t-j}} + (-1)^{t+1}\frac{(2t+2)^{\underline{t+1}}}{\vartheta+2}.\end{aligned} \tag{3.68}$$

This means that

$$P_t(\vartheta) = (\vartheta+2)\,\vartheta^{\underline{t}}\left(\sum_{j=0}^{t}(-1)^j(2t+2)^{\underline{j}}(\vartheta-j)^{\underline{t-j}}\right). \tag{3.69}$$

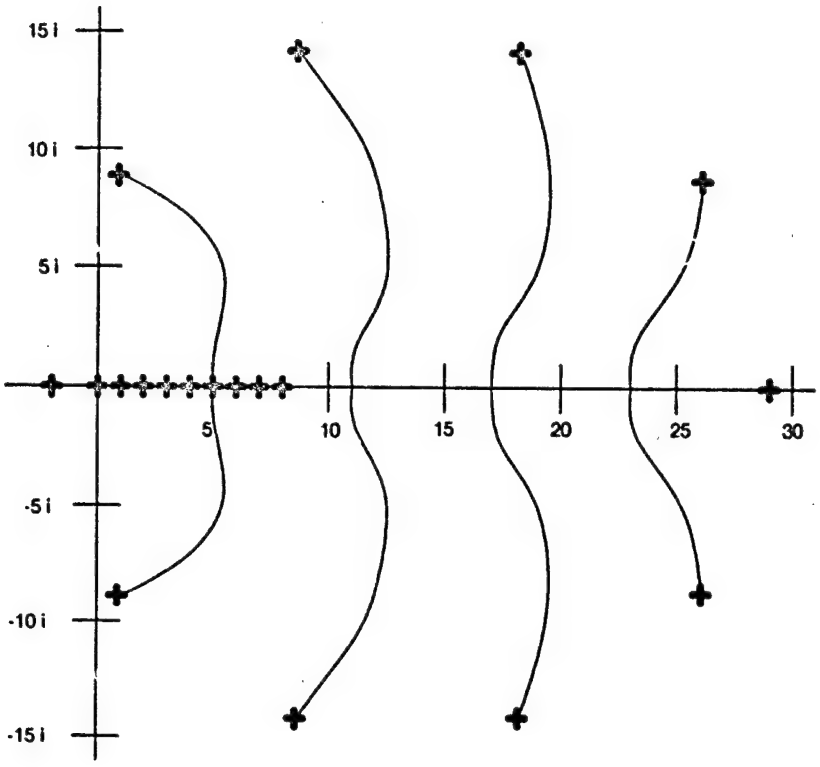


Figure 3.8 The Roots of $P_0(\vartheta)$

Several constants related to the $P_t(\vartheta)$ polynomials will also prove useful. Let

$$R_t(\vartheta) = \frac{P_t(\vartheta)}{\vartheta + 2}. \tag{3.70}$$

We wish to compute $R_t(-2)$ and $R_t'(-2)$ in order to apply equation (3.48). Using L'Hôpital's rule these quantities can be expressed in terms of $P_t(\vartheta)$:

$$\begin{aligned} R_t(-2) &= P_t'(-2) \\ R_t'(-2) &= \left.\frac{(\vartheta+2)P_t'(\vartheta) - P_t(\vartheta)}{(\vartheta+2)^2}\right|_{\vartheta=-2} \\ &= \left.\frac{(\vartheta+2)P_t''(\vartheta) + P_t'(\vartheta) - P_t'(\vartheta)}{2(\vartheta+2)}\right|_{\vartheta=-2} \\ &= \frac{P''(-2)}{2}. \end{aligned} \tag{3.71}$$

Since $P_t(\vartheta)$ is the difference of two falling factorials of ϑ (see equation (3.63)) we can compute these constants once we compute the derivative of a falling factorial:

$$\begin{aligned}
\vartheta^{\underline{j}} &= \left\langle \frac{z^j}{j!} \right\rangle (1+z)^\vartheta \\
\frac{\partial}{\partial \vartheta}\vartheta^{\underline{j}} &= \left\langle \frac{z^j}{j!} \right\rangle (1+z)^\vartheta \ln(1+z) \\
&= (-1)^{j+1}(H_{j-\vartheta-1} - H_{-\vartheta-1})(j-\vartheta-1)^{\underline{j}} \\
\frac{\partial^2}{\partial \vartheta^2}\vartheta^{\underline{j}} &= \left\langle \frac{z^j}{j!} \right\rangle (1+z)^\vartheta \ln(1+z)^2 \\
&= (-1)^j\left((H_{j-\vartheta-1} - H_{-\vartheta-1})^2 - (H^{(2)}_{j-\vartheta-1} - H^{(2)}_{-\vartheta-1})\right)(j-\vartheta-1)^{\underline{j}}.
\end{aligned} \tag{3.72}$$

For $\vartheta = -2$:

$$\begin{aligned}
\left.\frac{\partial}{\partial \vartheta}\vartheta^{\underline{j}}\right|_{\vartheta=-2} &= (-1)^{j+1}(H_{j+1}-1)(j+1)! \\
\left.\frac{\partial^2}{\partial \vartheta^2}\vartheta^{\underline{j}}\right|_{\vartheta=-2} &= (-1)^j\left((H_{j+1}-1)^2 - H^{(2)}_{j+1} + 1\right)(j+1)!.
\end{aligned} \tag{3.73}$$

Assembling these results gives the constants of interest:

$$\begin{aligned}
R_t(-2) &= (2t+2)!\,(H_{2t+2} - H_{t+1}) \\
R_t'(-2) &= \frac{(2t+2)!}{2}\,(H^2_{t+1} - H^2_{2t+2} + H^{(2)}_{2t+2} - H^{(2)}_{t+1}) + R_t(-2).
\end{aligned} \tag{3.74}$$

Returning to equation (3.62) for the number of nodes in a diminished data structure, the careful study of differential equations and $P_t(\vartheta)$ now bears fruit:

$$\begin{aligned}
P_t(\vartheta)\,T &= -\frac{(2t+1)!}{v} \\
T &= \frac{k}{R_t(-2)}v^{-2} - \frac{(2t+1)!}{P_t(-1)}v^{-1} + \cdots.
\end{aligned} \tag{3.75}$$

The constant k is our remaining concern. Recall that k appears after the first root, $\vartheta+2$, is factored from $P_t(\vartheta)$, leaving the equation

$$R_t(\vartheta)\,T = kv^{-2} - \frac{(2t+1)!}{v}. \tag{3.76}$$

We can use the initial conditions to compute $R_t(\vartheta)T|_{v=1}$ and thereby find a value for k. The first few terms of S,

$$S = c^2x^t + c^2x^{t+1} + \cdots c^2x^{2t} + \cdots, \tag{3.77}$$

give the first few terms of T:

$$T = 2x^t + 2x^{t+1} + \cdots + 2x^{2t} + \cdots . \tag{3.78}$$

So initially we know:

$$T^{(j)}(0) = \begin{cases} 0 & 0 \le j < t \\ 2j! & t \le j \le 2t \end{cases} . \tag{3.79}$$

Or, changing variables and using the ϑ operator

$$\vartheta^{\underline{j}} T|_{v=1} = \begin{cases} 0 & 0 \le j < t \\ (-1)^j 2j! & t \le j \le t \end{cases} . \tag{3.80}$$

By rewriting $R_t(\vartheta)$ slightly we have

$$R_j(\vartheta) = \sum_{j=0}^{t} (-1)^j (2t+2)^{\underline{j}}\, \vartheta^{\underline{2t-j}} \tag{3.81}$$

so that

$$\begin{aligned} R_t(\vartheta)\, T|_{v=1} &= \sum_{j=0}^{t} (2t+2)^{\underline{j}}\, 2(2t-j)! \\ &= 2(2t+2)! \sum_{j=0}^{t} \frac{1}{(2t+2-j)(2t+1-j)} \\ &= 2(2t+2)! \left(\frac{1}{t+1} - \frac{1}{2t+2} \right) \\ &= 2(2t+1)! . \end{aligned} \tag{3.82}$$

Solving for k:

$$\begin{aligned} k &= R_t(\vartheta)\, T|_{v=1} + (2t+1)! \\ &= 3(2t+1)! , \end{aligned} \tag{3.83}$$

gives a dominant term of

$$T = \frac{3}{2t+2} \frac{1}{H_{2t+2} - H_{t+1}} v^{-2} + \cdots \tag{3.84}$$

which expands with the binomial theorem:

$$\langle x^n \rangle\, T = \frac{3}{2t+2} \frac{1}{H_{2t+2} - H_{t+1}} (n+1) + o(n) . \tag{3.85}$$

So the number of allocations necessary to build a diminished tree is proportional to the number of keys and when t goes to infinity the allocations fall off inversely with t.

The $o(n)$ error term on equation (3.85) is necessarily weak, since this estimate depends on the other roots of $P_t(\vartheta)$ and some of these roots are migrating (as $t \to \infty$) towards points with -2 real part. In practice, however, the real part of these migrating roots is still positive when $t = 9$, as shown in Figure 3.8, and further analysis will suggest that we would never want an accumulator as large as 19, so we may safely use an error estimate of $O(1)$ in the above formula.

Another quantity of interest, the total memory usage, is closely related to the total number of nodes in the data structure. There are two approaches to the analysis. The more systematic approach uses a modification of the above grammar:

$$\begin{aligned} S &\to [B]\, m^3 x_{[a]} l\, S_{[fgh]} h l S_{[mno]} h \;\Big|\; m^3 y l h l m^4 A h \\ A &\to bxxe \;\Big|\; bxxxe \;\Big|\; bxxxxe\,. \end{aligned} \tag{3.86}$$

The amount of memory consumed appears in the exponents of the m's: 3 spaces for a tree node and 4 spaces for an accumulator. In general, an accumulator takes $2t$ spaces, although it is sometimes easier to program with $2t + 1$ spaces so that the accumulator can gracefully absorb $(2t + 1)$ keys before splitting.

The technique used here deserves highlighting. We have seeded the original grammar with a dummy variable, m, raised to the power of a quantity of interest, in this case the amount of memory used. We will repeatedly use this technique to study other quantities of interest, such as comparisons and memory probes.

The conversion and solution of grammar (3.86) is almost identical to the node counting grammar just solved, so it is omitted. The average amount of memory used to store n keys turns out to be:

$$\frac{t+3}{t+1}\,\frac{1}{H_{2t+2} - H_{t+1}}\,(n+1) + o(n)\,. \tag{3.87}$$

Comparing this with the $3n$ memory required for ordinary tree search, we see that the memory usage increases to $\frac{24}{7}n$ for $t = 1$, but then shows improvement for $t > 1$, converging to $\frac{1}{\ln 2}n$ as $t \to \infty$, a savings of 52%!

An alternate approach to this analysis relates nodes and memory usage, so that only one of the last two computations is necessary. A diminished tree will always have an odd number of tree nodes, call this $2j - 1$, of which j are dummy nodes and $j - 1$ are internal nodes. The j dummy nodes have j accumulators as right descendants. We can compute the total number of nodes, $nodes = 3j - 1$, and the total amount of memory, $memory = (6 + 2t)j - 3$, and then relate the two quantities:

$$memory = \frac{(6+2t)}{3}\,(nodes + 1) - 3\,. \tag{3.88}$$

Since this is a linear relationship, the mean and variance of these two quantities are similarly related.

The next analysis presents fresh difficulties, but it also answers a critical question about the usefulness of the new data structure: on the average, how many comparisons are required to find a key in a diminished tree? This involves studying path length in much the same way we studied paths in ordinary trees, using an index on the nonterminals to record their depth within the tree:

$$\begin{aligned} S_k &\to [B]q^k x_{\,a}\, lS_{k+1[fgh]}\, hlS_{k+1[mno]}h \;\big|\; ylhlA_k h \\ A_k &\to bq^{k+1}xq^{k+2}xc \;\big|\; bq^{k+1}xq^{k+2}xq^{k+3}xc \;\big|\; bq^{k+1}xq^{k+2}xq^{k+3}xq^{k+4}xe. \end{aligned} \tag{3.89}$$

Each x in the tree is prefixed with q raised to a power equal to the number of comparisons necessary to find the key stored at that location. The total number of q's in a tree divided by the number of keys is the average successful search time for the tree.

The grammar converts to an equation,

$$\begin{aligned} S_k = q^k \binom{2t+1}{t}(t+1)\int^{(2t+1)} (S_{k+1}^{(t)})^2 \\ + q^{tk+\binom{t+1}{2}}x^t + q^{(t+1)k+\binom{t+2}{2}}x^{t+1} + \cdots + q^{2tk+\binom{2t+1}{2}}x^{2t}, \end{aligned} \tag{3.90}$$

where S_k is a function of x and q. Applying the Q operator to this equation we find that $QS_k = S_{k-1}$, so by the fixed point theorem of Chapter 1, S_1 can be expressed as a function of itself:

$$\begin{aligned} S_1 = q\binom{2t+1}{t}(t+1)\int^{(2t+1)} ((QS_1)^{(t)})^2 \\ + q^{t+\binom{t+1}{2}}x^t + q^{t+1+\binom{t+2}{2}}x^{t+1} + \cdots + q^{2t+\binom{2t+1}{2}}x^{2t}. \end{aligned} \tag{3.91}$$

This has a differential form

$$S_1^{(2t+1)} = q\binom{2t+1}{t}(t+1)q^{2t}Q(S^{(t)})^2 \tag{3.92}$$

with initial solution:

$$S_1 = q^{\binom{t+2}{2}-1}x^t + q^{\binom{t+3}{2}-1}x^{t+1} + \cdots + q^{\binom{2t+2}{2}-1}x^{2t} + \cdots. \tag{3.93}$$

To find the mean of the total path length we let

$$T = \frac{\partial}{\partial q} S_1 \bigg|_{q=1}, \tag{3.94}$$

and then differentiate equation (3.92) with respect to q and set $q = 1$.

$$T^{(2t+1)} = \binom{2t+1}{t}(t+1)\Big((2t+1)(S^{(t)})^2 + 2S^{(t)}(xS^{t+1} + T^{(t)})\Big). \tag{3.95}$$

Here we have used the chain rule to differentiate $Q\,S^{(t)}$. Once q is set to one S_1 becomes $S = 1/(1-x)$ and can be replaced using the formula:

$$S^{(t)} = \frac{t!}{(1-x)^{t+1}}. \tag{3.96}$$

This yields

$$(1-x)^{2t+1}T^{(2t+1)} - 2(2t+1)^{\underline{t+1}}(1-x)^t T^{(t)} = (2t+1)!\,\frac{2t+1}{1-x} + (2t+1)!\,\frac{2(t+1)x}{(1-x)^2}. \quad (3.07)$$

And with the substitutions $v = 1 - x$, and $\vartheta = v\frac{\partial}{\partial v}$ we find a familiar form of differential equation with a familiar polynomial $P_t(\vartheta)$:

$$\begin{aligned} P_t(\vartheta)\,T &= -(2t+1)!\left(\frac{2t+1}{v} + \frac{2(t+1)(1-v)}{v^2}\right) \\ &= -(2t+1)!\left(\frac{2t+2}{v^2} - \frac{1}{v}\right) \qquad (3.08) \\ P_t(\vartheta) &= \vartheta^{\underline{2t+1}} + (-1)^t 2(2t+1)^{\underline{t+1}}\vartheta^{\underline{t}}. \end{aligned}$$

This time however there is an inhomogeneous term, $-(2t+2)!\,v^{-2}$, with an exponent that matches the dominant root $\vartheta = -2$ of $P_t(\vartheta)$, a situation dealt with in equation (3.48). This gives a leading term of

$$T = -(2t+2)!\,\frac{v^{-2}\ln v}{R_t(-2)} + \cdots. \qquad (3.09)$$

Replacing v with $1 - x$ and applying Zave's identity yields

$$\langle x^n\rangle\,T = \frac{1}{H_{2t+2} - H_{t+1}}\,(H_{n+1} - 1)(n+1) + o(n). \qquad (3.100)$$

Comparing this with the leading term of equation (3.28), $2(H_{n+1} - 1)(n-1)$, we see that when $t = 1$, diminished searching already shows an advantage (12/7 versus 2 in the constant) over ordinary searching. For large n this is a 14% improvement in the number of comparisons necessary for successful searching. As t grows large the leading constant converges to $1/\ln 2$, a 28% improvement in the number of comparisons.

However, we must not rejoice prematurely since the next, linear, term in the expansion can have a dramatic effect on the usefulness of the algorithm. Compare

$$2N\ln N + 2N \qquad (3.101)$$

with

$$\frac{12}{7}N\ln N + 6N. \qquad (3.102)$$

While the second formula is asymptotically 14% smaller than the first, the break even point is larger than a million. Fortunately the break even point for diminished searching is considerably smaller, but in order to satisfactorily analyze the algorithm we must compute the constant in the second term of the expansion.

To obtain this constant we return to the solution of the differential equation. After $\vartheta + 2$ is factored from the equation there is an undetermined constant, k, in the remaining equation:

$$R_t(\vartheta)\,T = -(2t+2)!\,\frac{\ln v}{v^2} + \frac{k}{v^2} + \frac{(2t+1)!}{v}. \qquad (3.103)$$

When the other roots are removed, k remains in the solution:

$$T = \frac{-(2t+2)!}{R_t(-2)}\frac{\ln v}{v^2} + (2t+2)!\,\frac{R_t'(-2)}{R_t(-2)^2}\frac{1}{v^2} + \frac{k}{R_t(-2)}\frac{1}{v^2} + \cdots. \tag{3.104}$$

The constant k is found by setting $v = 1$ in equation (3.103) and computing $R_t(\vartheta)T|_{v=1}$ from the initial conditions. Based on equation (3.93) the first few terms of T are:

$$T = \left(\binom{t+2}{2} - 1\right)x^t + \left(\binom{t+3}{2} - 1\right)x^{t+1} + \cdots + \left(\binom{2t+2}{2} - 1\right)x^{2t}. \tag{3.105}$$

This gives initial conditions of

$$T^{(j)}(0) = \begin{cases} 0 & 0 \le j < t \\ j!\left(\binom{j+2}{2} - 1\right) & t \le j \le 2t, \end{cases} \tag{3.106}$$

or

$$\vartheta^{\underline{j}}T|_{v=1} = \begin{cases} 0 & 0 \le j < t \\ (-1)^j j!\left(\binom{j+2}{2} - 1\right) & t \le j \le 2t \end{cases}. \tag{3.107}$$

We can now compute

$$\begin{aligned} R_t(\vartheta)T|_{v=1} &= \sum_{j=0}^{t}(-1)^j(2t+2)^{\underline{j}}\vartheta^{\underline{2t-j}}T|_{v=1} \\ &= \sum_{j=0}^{t}(2t+2)^{\underline{j}}(2t-j)!\left(\binom{2t-j+2}{2} - 1\right) \\ &= (2t+2)!\left(\frac{t+1}{2} - \frac{1}{2t+2}\right). \end{aligned} \tag{3.108}$$

So k is given by

$$\begin{aligned} k &= (2t+2)!\left(\frac{t+1}{2} - \frac{1}{2t+2}\right) - (2t+1)! \\ &= (2t+2)!\left(\frac{t+1}{2} - \frac{1}{t+1}\right). \end{aligned} \tag{3.109}$$

Then by applying Zave's identity to equation (3.104), and using the results $R_t(-2)$ and $R_t'(-2)$ computed already, we get the desired solution:

$$\begin{aligned} \langle x^n\rangle T = {} & \frac{1}{H_{2t+2} - H_{t+1}}(H_{n+1} - 1)(n+1) + \\ & \frac{1}{H_{2t+2} - H_{t+1}}\left(\frac{t+1}{2} - \frac{H_{2t+2}}{2} - \frac{H_{t+1}}{2} + 1 + \frac{1}{2}\frac{H_{2t+2}^{(2)} - H_{t+1}^{(2)}}{H_{2t+2} - H_{t+1}} - \frac{1}{t+1}\right)(n+1) + o(n). \end{aligned} \tag{3.110}$$

The table below summarizes these constants for small values of t. The break even column on the right shows when an accumulator of size t proves better than an accumulator of size $t-1$. For $t=1$, the algorithm performs better than ordinary tree search once n exceeds 20.

Half Accumulator t	Coefficient of $(H_{n+1}-1)(n+1)$	Coefficient of $(n+1)$	Break even n t with $t-1$
No accumulator	2	-1	—
1	$\frac{12}{7}\sim 1.714$	$-\frac{12}{49}\sim -.245$	20
2	$\frac{60}{37}\sim 1.622$	$\frac{290}{1369}\sim .212$	210
3	$\frac{840}{533}\sim 1.576$	$\frac{193060}{284089}\sim .680$	4.3×10^4
4	$\frac{2520}{1627}\sim 1.549$	$\frac{3118374}{2647129}\sim 1.178$	1.5×10^8

Many quantities of interest can be analyzed in a way similar to the analysis just completed. For example, perhaps the most realistic estimate of the cost of successful searching is the number of memory references made by the algorithm. The diminished tree algorithm wastes 3 memory references on the dummy nodes but then recoups some of this loss on the linear, one probe per key, search of the accumulator lists. If we assume that one probe is necessary to find the root of the tree, then the appropriately weighted grammar is:

$$\begin{aligned} S_k &\to [B]q^{2k}z_{[a]}lS_{k+1[fgh]}hlS_{k+1[mno]} \mid ylhlA_kh \\ A_k &\to bq^{2k+3}xq^{2k+4}xc \mid bq^{2k+3}xq^{2k+4}xq^{2k+5}xe \mid bq^{2k+3}xq^{2k+4}xq^{2k+5}xq^{2k+6}xe. \end{aligned} \tag{3.111}$$

This time the infinite set of equations is resolved with a double application of the Q operator, $S_{k+1}=Q^2S_k$. The average number of probes for a tree of size n turns out to be:

$$\begin{aligned} &\frac{2}{H_{2t+2}-H_{t+1}}(H_{n+1}-1)(n+1) \\ &+\frac{1}{H_{2t+2}-H_{t+1}}\left(\frac{t+1}{2}+2H_{2t+2}-4H_{t+1}+2\frac{H^{(2)}_{2t+2}-H^{(2)}_{t+1}}{H_{2t+2}-H_{t+1}}-\frac{3}{t+1}\right)(n+1)+o(N). \end{aligned} \tag{3.112}$$

To study comparisons in unsuccessful searching, we insert q's between each of the nodes in the accumulator lists. Any new key will land in one of these slots, and the exponent of the q will encode the number of comparisons used to locate the correct slot.

$$\begin{aligned} S_k &\to [B]xl\,S_{k+1}hl\,S_{k+1} \mid ylhlA_kh \\ A_k &\to bq^{k+1}xq^{k+2}xq^{k+2}c \mid bq^{k+1}xq^{k+2}xq^{k+3}xq^{k+3}c \mid bq^{k+1}xq^{k+2}xq^{k+3}xq^{k+4}xq^{k+4}e. \end{aligned} \tag{3.113}$$

This set of equations is resolved with the operator qQ, and the generating function has coefficients

$$\frac{1}{H_{2t+2}-H_{t+1}}(H_{n+1}-1)(n+1)+$$
$$\frac{1}{H_{2t+2}-H_{t+1}}\left(\frac{t+1}{2}+\frac{1}{2}H_{2t+2}-\frac{3}{2}H_{t+1}+1-\frac{1}{2}\frac{H^{(2)}_{2t+2}-H^{(2)}_{t+1}}{H_{2t+2}-H_{t+1}}-\frac{1}{2t+2}\right)(n+1). \tag{3.114}$$

Dividing by $n+1$ gives the average number of comparisons in an unsuccessful search. Equation (3.114) compares favorably with a similar quantity for ordinary tree search:

$$2(H_{n+1}-1)(n+1). \tag{3.115}$$

However, the most interesting quantities for unsuccessful searching are the memory reads and writes, especially in cases where the accumulator lists are splitting. Treating writes separately from reads, we obtain a grammar for writes:

$$\begin{aligned} S_k &\to [B]xlS_{k+1}hlS_{k+1} \mid ylhlA_kh \\ A_k &\to bw^3xw^2xwe \mid bw^4xw^3xw^2xwc \mid bw^{5+11}xw^{4+11}xw^{3+11}xw^{2+11}xw^{1+11}e. \end{aligned} \tag{3.116}$$

Most of the w exponents correspond to moving segments of the accumulator lists and writing the new key. But when the accumulator is full there is an extra cost for creating new tree nodes and copying half the old accumulator into a new accumulator. This is 11 in the above formula, and will be $t+9$ in general (t for copying half the accumulator, 6 for writing the fields of two new dummy nodes, and 3 for rewriting the old dummy node and its pointers so that it contains the median key). Solving (3.116) yields a generating function with coefficients equal to the average number of writes:

$$\frac{1}{H_{2t+2}-H_{t+1}}\left(\frac{t+1}{2}+\frac{1}{2}+\frac{4}{t+1}\right)(n+1). \tag{3.117}$$

This is slightly worse than the 4 writes required for ordinary tree search.

Memory reads are set up as follows:

$$\begin{aligned} S_k &\to [B]xlS_{k+1}hlS_{k+1} \mid ylhlA_kh \\ A_k &\to bq^{2k+4}xq^{2k+4}xq^{2k+4}e \mid bq^{2k+5}xq^{2k+5}xq^{2k+5}xq^{2k+5}e \mid \\ &\qquad bq^{2k+6+3}xq^{2k+6+3}xq^{2k+6+3}xq^{2k+6+3}xq^{2k+6+3}e. \end{aligned} \tag{3.118}$$

In most cases the algorithm will use $2k+2$ reads to reach the accumulator, and then it will read the entire accumulator, either to find the proper location for the new key, or to move the keys that were not searched. When the accumulator is full, there is extra reading (3 probes in equation (3.118), and $t+1$ in general) to remove the median and the larger half of the accumulator.

This family of equations is resolved with q^2Q^2, and solved with the usual techniques:

$$\frac{2}{H_{2t+2}-H_{t+1}}(H_{n+1}-1)(n+1)+$$
$$\frac{1}{H_{2t+2}-H_{t+1}}\left((t+1)+H_{2t+2}-3H_{t+1}+\frac{5}{2}+\frac{H^{(2)}_{2t+2}-H^{(2)}_{t+1}}{H_{2t+2}-H_{t+1}}\right). \tag{3.119}$$

This is to be compared with

$$4(H_{n+1}-1)(n+1)+(n+1), \tag{3.120}$$

for ordinary tree search. To get the average number of memory reads we divide these equations by $n+1$.

Without knowing the exact costs of various machine operations and the nature of the compiler used it is difficult to make a precise recommendation for the use of diminished searching. The figures below summarize break even points for some of the factors likely to affect the problem. They are based entirely on memory reads and writes and the assumption that they both cost one unit of time.

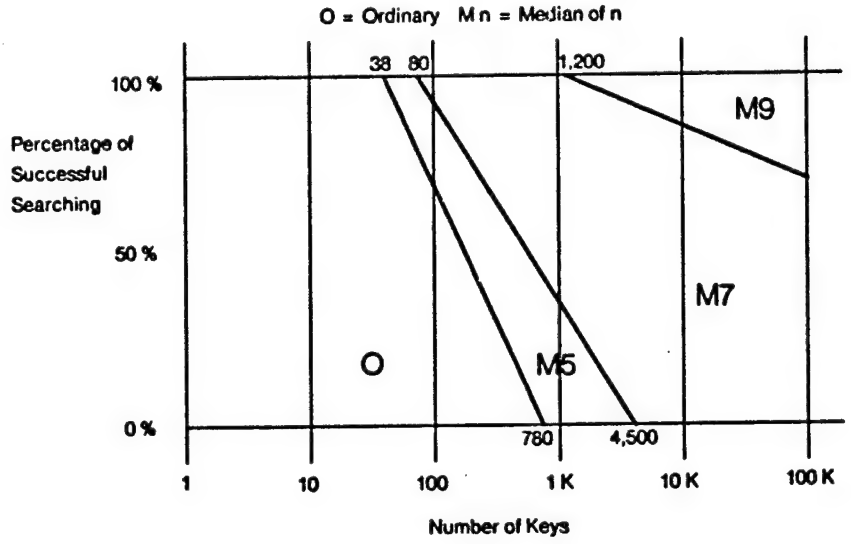


Figure 3.9 Break Even Points and the Ratio of Successful and Unsuccessful Searching

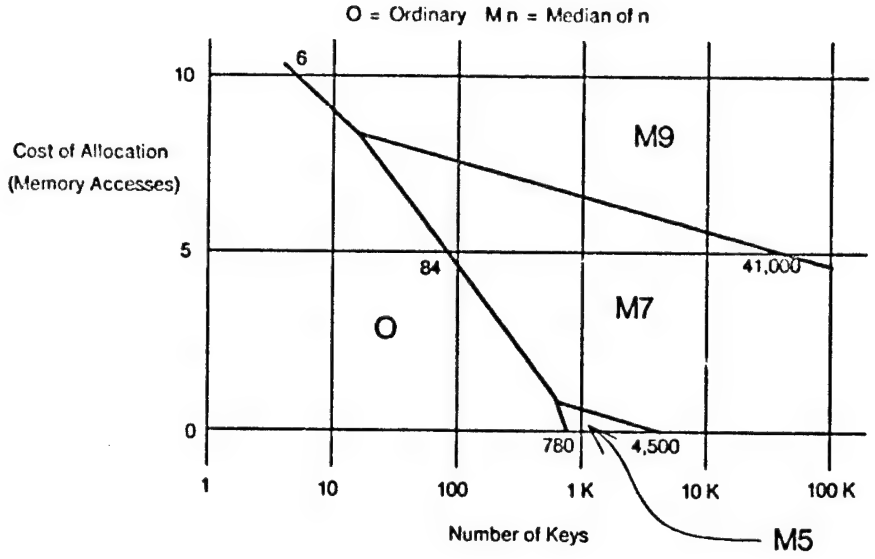


Figure 3.10 Break Even Points and the Cost of Node Allocation

Of course the exact choice of algorithm depends on a combination of the equations of this section with appropriate cost estimates. Nevertheless, these figures give us a good perspective on the design space of the problem. Notice that median of 3 is completely skipped; it is better to use either median of 5, or ordinary tree search. This jump was anticipated by equation (3.87), showing that total memory use increased for median of 3; it appears that median of 5 is better for both access time and memory use.

The designer should be particularly sensitive to the way memory is allocated, since this can significantly affect the choice of t. With a simple stack allocator the best choice of t is usually 2 (median of 5). If a complicated memory manager is used, then larger values of t will greatly reduce the number of allocations (see equation (3.85)), decreasing the constant on the linear term, and shifting the break even points. In the region of 1000 keys, a choice of t equal to 3 or 4 becomes attractive.

Finally it is worth noting two features not shown in Figures 3.9 and 3.10. These figures show break even points between algorithms—not points of substantial advantage. To obtain a point where half the asymptotic improvement is realized, a good rule of thumb is to square the break even point.

The figures also ignore improvements due to block memory access. Many larger systems fetch blocks of 4-8 words from main memory, and have fast instructions for moving blocks of memory. Both of these features harmonize well with diminished tree searching.

CHAPTER 4

GENERATION AND RECOGNITION

While the previous chapter used labelled grammars to analyze algorithms that were, at least on the surface, unrelated to labelled formal languages, this chapter is devoted to algorithms that operate directly on the grammars. Since a grammar describes a family of combinatorial objects, it is reasonable to ask all the standard questions about that family: Can we generate a member of the family uniformly at random? Can we generate all members of the family? Except we ask these questions in a more general setting: Can we build a system to accept an arbitrary grammar and then generate all objects, or generate objects uniformly at random from the specified family?

Such a general-purpose system is possible. It takes as input a labelled grammar, and then provides a number of functions related to the grammar. These functions fall into two categories. The *generation* category includes:

1) Computing the size of a specified family of combinatorial objects.

2) Selecting an element by rank within such a family.

3) Generating an element uniformly at random from the family.

4) Generating all elements in the family.

The specification of a family has two parts. The grammar provides what might be called a *shape* description; it could, for example, constrain the strings to represent permutations in cycle format:

$$\begin{aligned} P &\rightarrow C^{\square} bP \mid \epsilon \\ C &\rightarrow x^{\square} R \\ R &\rightarrow xR \mid \epsilon . \end{aligned} \tag{4.1}$$

The user must also provide a *size* description, that is, counts of critical characters in the string. For this grammar the user might specify 3 b's and 6 x's. With both a *shape* and a *size* description a general system can count the family members (there are 225 permutations of 6 elements with 3 cycles) or generate instances at random (such as $x_1x_2x_6bx_3x_5bx_4b$).

Hereafter, we will refer to those characters whose occurrences are of interest as *critical characters*. In the last example x and b are critical. The number of critical characters is the *dimension* of the problem, and the counts of critical characters in the size description form a *characteristic vector*. By convention, the count of the labelled character will appear

in the first component of the vector. Returning again to the above example, we have a two dimensional characteristic vector (6, 3).

The second, *recognition*, category is, in a loose sense, the inverse of the generation category. It includes:

1) Testing if a string can be derived with a grammar.

2) Ranking a string within a specified family.

For example, a tester would reject $x_3x_1bx_2b$ since it cannot be derived with the above grammar, while a ranker might give $x_1x_3bx_2b$ a rank of 2 within the family of permutations on three elements having two cycles. The rank has no absolute meaning; we do, however, expect that each member of the family will have a different rank, and that the ranking function of this category is an inverse of the selection function of the preceding category.

These generation and recognition problems have received considerable attention in the literature. There are special purpose algorithms for various combinatorial objects, and general frameworks for large classes of problems [Nijenhuis 1978] [Wilf 1977 and 1978] [Williamson 1976]. The interest stems from numerous applications: Hypotheses can be tested with complete searches. Algorithms can be studied empirically with random input. And selection and ranking algorithms can serve as perfect hash functions.

The general-purpose algorithms of this chapter capitalize on the close connection between labelled grammars and generating functions. Since the algorithms are closely related, we will begin with the simple counting problem and then explore the difficulties presented by more complex functions. The descriptions below are restricted to labelled formal languages with the box operator. All the algorithms generalize to more complicated partial orders, but the details would obscure the basic features.

For purposes of exposition it is also helpful to restrict the grammars so that the right sides of all productions have either two nonterminals or a single terminal. The start nonterminal is the only nonterminal that can derive an empty string, and in these cases, the start nonterminal cannot appear on the right side of any production. For ordinary formal languages, a grammar in this restricted format is considered to be in Chomsky normal form. There are algorithms to remove ϵ-productions, eliminate chains like $A \to B$, $B \to C$, and $C \to D$, and break up larger productions that will expand the grammar by no more than squaring the number of productions, and will not introduce ambiguity into an unambiguous grammar. (See [Harrison 1978; Chapter 4].) These same algorithms will work for labelled grammars as long as we take some care when we break up larger productions with the box operator. The production $P \to QRS^{\square}TU$, for example, is equivalent to

$$\begin{aligned} P &\to QP_1^{\square} \\ P_1 &\to RP_2^{\square} \\ P_2 &\to S^{\square}P_3 \\ P_3 &\to TU\,. \end{aligned} \tag{4.2}$$

In general, we can decompose a large production into any binary tree of two nonterminal productions, and then box every nonterminal on the path through the tree to the boxed

element of the original production. If boxes are handled this way, then the traditional algorithms for converting arbitrary context free grammars into Chomsky normal forms will also normalize labelled grammars.

4.1 Generation Problems

4.1.1 Counting

Chapter 2 developed the direct correspondence between labelled grammars and integral equations in order to solve the equations for closed form generating functions, or at least to recover information analytically from the equations. In this section we are interested in solving these equations mechanically for specific terms of the series expansion. This can be done by iterating the equations (in a carefully chosen order) until the desired solution is reached, but before we explore the method of iteration, we need to understand the basic computation step. Suppose that the nonterminals of the grammar are $P, Q, R, \ldots$, and that the terminal characters x, a, and b are critical (x being the special labelled character). Each nonterminal will have a three dimensional series expansion:

$$P(x,a,b) = \sum_{j,k,l} P_{jkl} \frac{x^j}{j!} a^k b^l , \tag{4.3}$$

with integer coefficients in a matrix P associated with the nonterminal. The (j,k,l) entry in P is the number of different strings derivable from P having j x's, k a's, and l b's. A production like $P \to QR$ indicates that the convolution of the Q and R matrices will contribute to the terms in P:

$$P_{jkl} \leftarrow P_{jkl} + \sum_{m,n,o} \binom{j}{m} Q_{mno} R_{j-m,k-n,l-o} . \tag{4.4}$$

Notice that the convolution has an extra factor $\binom{j}{m}$. Hereafter this will be called the *split factor*, since it accounts for the splitting of labels between the nonterminals. In general, the split factor depends on the box operator. If Q is boxed in the above production then the factor is $\binom{j-1}{m-1}$, or if R is boxed, the factor is $\binom{j-1}{m}$. Boxing removes one from the corresponding term (m or $j-m$) in the binomial coefficient.

Equation (4.4) above highlights one of the difficulties of computation. P_{jkl} can depend on most of the matrix entries in Q and R, so we must compute all the matrices together. One way to avoid conflict is to compute the entries in an order that always increases the sum of their components. Every location (j,k,l) with $j+k+l = t$ is computed before any location with $j+k+l = t+1$. However, this order does not resolve all the problems that can arise. Returning again to equation (4.4), if we have Q_{000} nonzero, then R_{jkl} should be computed before P_{jkl}. So even within a particular location we must follow a special, *safe*, order of computation among the nonterminals.

To find the *safe* order we identify those nonterminals, like Q_{000} above, that can derive strings that are free of critical characters. Such strings will be be called *pseudo* empty. Algorithms that identify nonterminals that derive pseudo empty strings are well known,

although they appear in the guise of computing functional dependencies [Ullman 1980], or testing for emptiness of a language [Aho 1972].

Definition. *A safe order of computation is a linear embedding of the following partial order:* $N_i \prec N_j$ *(read* N_i *precedes* N_j*) if there is a production* $N_j \to N_i N_k$ *or* $N_j \to N_k N_i$ *and* N_k *derives a pseudo empty string.*

If there is a cycle in the partial order, e.g. $N_i \prec N_j \prec N_k \prec N_i$, then the count is ill defined; the cycle generates an infinite number of derivations with a fixed number of occurrences of the critical characters.

Altogether, we have described the following, iterative, approach to computing the nonterminal matrices:

```
for t ← 0 to ∞ do
  for j ← 0 to t do
    for k ← 0 to t − j do
      begin
      l ← t − j − k;
      for each nonterminal P according to the safe partial order do
        compute P_jkl using equation (4.4);
      end
```

The running time of this algorithm can be expressed in terms of

d	The number of characters critical to the counting
n	The number of nonterminals
m	The maximum number of occurrences of any given critical character.

For most applications d and possibly n are small fixed constants, while m grows large and has the greatest impact on the space and running time of the algorithm.

The parameter d is the dimensionality of the problem. Each nonterminal has a d-dimensional array of coefficients, with m being the maximum coordinate in any dimension. Thus the space required is $O(nm^d)$.

The time necessary to fill the arrays is $O(nm^{2d})$ since each of the $O(nm^d)$ entries can require an $O(m^d)$ convolution of two series whenever there is a production with two nonterminals on the right side.

4.1.2 Selection and Generation at Random

The next step in difficulty, beyond counting derivations, is to produce strings derived by a grammar. If we anticipate that the typical user will first specify a family of objects with a grammar, and then make numerous requests for random or selected members of the family, then our efforts should be directed at streamlining the latter process. For this reason the exposition of the section is reversed. The section begins with the description of a *walk structure* that is well adapted to random generation and selection problems. Every combinatorial family has its own walk structure and this is used repeatedly and efficiently to generate strings. Only later in the section will it become clear how to create a walk structure from a grammar.

Labelled strings are generated by a *walk* procedure that traverses the data structure. The procedure has two arguments, an integer c specifying the string (or substring) to be generated, and a list of labels that will appear on the special characters of the generated string. For the first call to *walk*, c is specified by the user (in the case of a selection request) or generated at random. We initialize the label list with $1 \ldots j$ where j is the number of special characters expected in the output.

The walk structure is a binary tree with two types of nodes: trivial and drastic. All nodes contain an integer v, specifying the number of different traversals that can originate at the node, and possibly two pointers to descendant nodes. Trivial nodes are depicted as diamonds in the figures that follow.

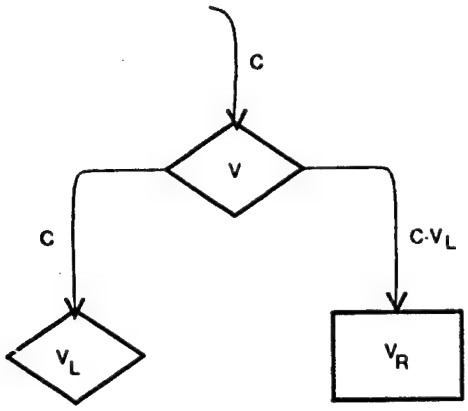


Figure 4.1 Trivial Nodes

When the traversing procedure reaches a trivial node, the integer carried by the procedure, c, should be in the range $0 \ldots v-1$. A comparison of c with the value of the left descendant, v_l, determines the direction of the walk: if $c < v_l$ the procedure traverses the left descendant, otherwise c is replaced by $c - v_l$ and the right descendant is traversed.

A drastic node can also have two offspring, but in this case both descendants are traversed. Each drastic node corresponds to firing a production in the original grammar, so there is a production right hand side such as a or QR associated with the node. The

traversing procedure will either output the terminal character a, or traverse both descendants to produce substrings for Q and R, in which case there are two additional fields l_l and l_r in the drastic node; they are the number of labels given to Q and R respectively.

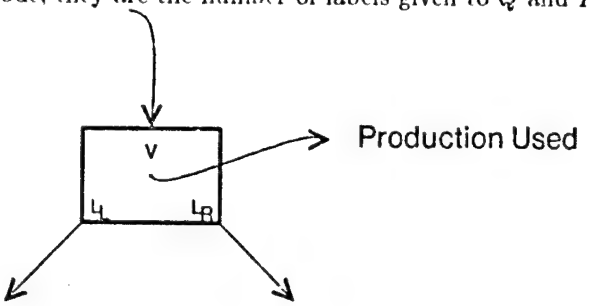


Figure 4.2 A Drastic Node

Recall that when j special characters are expected in the output, the top level call to the traversal procedure begins with a linked list of 1 through j. At a drastic nodes with descendants, the procedure splits the current list of labels into two lists for the descendents. If a box operator is applied to either nonterminal then the smallest label is moved to the designated group before the splitting process.

At a drastic node the integer c carried by the traversing procedure is divided into three integers, one for each descendant, and one to control the splitting of the labels:

$$\begin{aligned} c_r &\leftarrow c \bmod v_r \\ c &\leftarrow c \operatorname{div} v_r \\ c_l &\leftarrow c \bmod v_l \\ s &\leftarrow c \operatorname{div} v_l . \end{aligned} \tag{4.5}$$

Using that portion allocated for splitting, s, a list of labels can be partitioned into two groups of sizes a, and b:

```
while a + b > 0 do
  if s < (a-1+b choose a-1) then
    begin
    add the next label to the a list;
    a ← a − 1;
    end
  else
    begin
    s ← s − (a-1+b choose a-1);
    add the next label to the b list;
    b ← b − 1;
    end
```

The loop preserves the invariant that $a + b$ labels remain to be processed, and $0 \le s < \binom{a+b}{a}$. Since $\binom{a+b}{a} = \binom{a-1+b}{a-1} + \binom{a+b-1}{a}$ we can partition the range $\left[0, \binom{a+b}{a}\right)$ into two parts, one corresponding to putting the first label into the a group, in which case the remaining problem is to partition $a - 1 + b$ labels into groups of size $a - 1$ and b, and s is appropriately in the range $\left[0, \binom{a-1+b}{a-1}\right)$. Larger values of s correspond to putting the first label in the b group, and so s will be in the appropriate range after subtracting $\binom{a-1+b}{a-1}$. In this way the invariant is preserved.

We now have all the ingredients for the walk procedure:

```
procedure walk(tree, c, labels)
  case tree↑.node_type of
    trivial:
      if c < tree↑.left↑.value then
        call walk(tree↑.left, c, labels)
      else
        call walk(tree↑.right, c − tree↑.left↑.value, labels);
    drastic:
      begin
      if the production is of the form P → a then output a
      else
        begin
        use (4.5) to separate c into c_l, c_r and s;
        if a box appears in the production then
          remove the smallest integer from labels;
        use s to split labels into left labels and right labels;
        if a box appears in the production then
          return the smallest label to the appropriate group;
        call walk(tree↑.left, c_l, left labels);
        call walk(tree↑.right, c_r, right labels);
        end
```

So the *walk* procedure is a straightforward hybrid of tree search and tree traversal. Trivial nodes are "searched" and drastic nodes are "traversed" in order to generate labelled terminal strings.

Of course an obvious question remains: how do we construct a walk structure for a given family of combinatorial objects? The answer is surprisingly simple; we use the algorithm of the preceding section for counting the number of objects and replace the additions and multiplications in the computation with *structural* additions and multiplications that patch together pieces of walk structure.

A trivial node results from the addition of two nodes, either drastic or trivial. The value field of the new node is the sum of the value fields of the combined nodes:

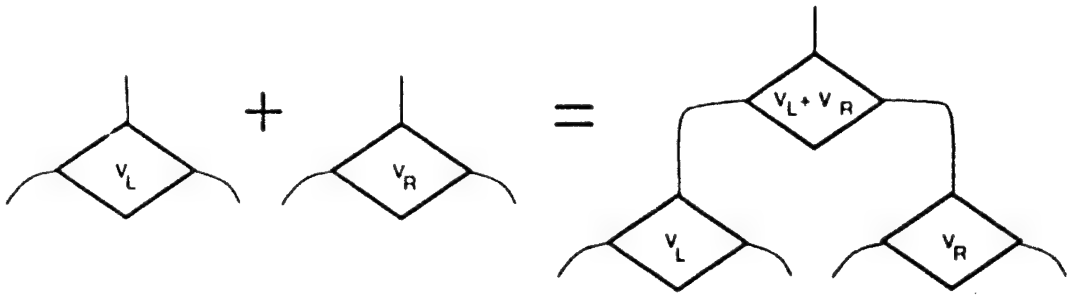


Figure 4.3 Summing Two Nodes

A drastic node is the result of the multiplication of two nodes:

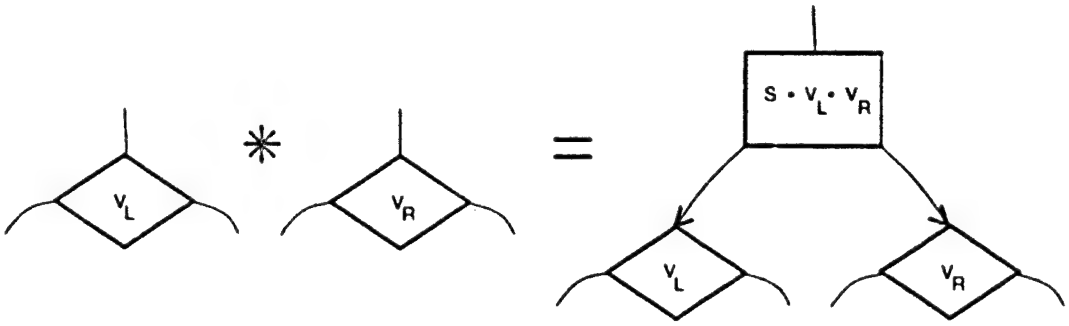


Figure 4.4 Multiplying Two Nodes

Most of the fields of the drastic node (such as the production used and the number of labels going left and right in the split) are provided by the context of the multiplication. The split factor S is computed according to the number of labels going left and right, and the value field of the new node is the product of S and the values of the two descendants.

In this way, the counting procedure of the preceding section can be adapted to compile a walk structure. To improve the walk structure, it is advantageous to postpone summing the nodes together as described above. Instead of immediately creating a trivial node whenever a sum occurs in the counting procedure, we first collect all those nodes that will eventually be part of the same sum. Once the sum is complete, a *balanced* walk structure can be built from the collected nodes for guaranteed logarithmic search time on a single search, or Huffman's algorithm [Knuth 1973; Section 2.3.4.5] can be used to build a walk structure that is optimal when averaged over all possible searches.

The running time and space requirements for the selection and generation at random algorithms fall into two classes: pre-computation to create a walk structure from a grammar, and post-computation to generate objects using the walk structure. The cost of pre-computation is based on the counting algorithm described earlier. The time required remains $O(nm^{2d})$, and because each multiplication and addition creates new data structure, the space required is also $O(nm^{2d})$.

Post-computation requires a bit more reasoning: since each drastic node corresponds to the firing of a production in the grammar and since each traversal generates a string of the grammar, the traversal will visit as many drastic nodes as there are productions contributing to the final string. If the final string is of length l, then we can use at most l productions generating terminal symbols, and at most l productions of the form $P \rightarrow QR$, since each nonterminal must generate at least one terminal. So the string generation can require at most $O(l)$ productions, and the traversal can visit at most $O(l)$ drastic nodes.

The traversal of a drastic node involves a splitting operation on a label list ($O(m)$ time) and the time necessary to find the next drastic node. Between drastic nodes there is a network of trivial nodes produced by the summation in the counting algorithm. Since there are at most $O(m^d)$ terms summed together (the convolution of two d-dimensional series) we expect to visit at most $O(m^d)$ trivial nodes, or if the balancing technique described above is used, $O(d \ln m)$ trivial nodes.

Altogether the time necessary to generate a string is $O(l(m + d \ln m))$.

4.1.3 Enumeration

Our next and last problem within the generation category is the systematic enumeration of all members of a combinatorial family. For a simple solution to this problem we can use the selection algorithm of the preceding section, sequencing the algorithm through all possible values. However, since most of the differences between the jth and $(j + 1)$st members of combinatorial families are confined to small regions of their strings, it is senseless to completely regenerate the strings. The unnecessary regeneration is avoided by loading the generated characters into a buffer, and then reusing all but the right end of the buffer for subsequent strings.

In the preceding section we developed a *walk* algorithm that produced a single string from an integer input. In this section the algorithm that enumerates all strings is called *all-walks*; it will occasionally use the *walk* algorithm as a subroutine.

All-walks uses the data structure compiled for walking, but it treats the nodes differently. At a trivial node the *all-walks* procedure calls itself recursively on both descendants (recall that the *walk* procedure chooses only one descendant). At a drastic node the procedure uses a set of nested loops to cover all possibilities:

if *the production is of the form* $P \rightarrow a$ **then** *append* a *to the buffer, and output the buffer*
else *the production is of the form* $P \rightarrow QR$, *so ...*
for *all possible splits of the label list* **do**
  begin
  for $n \leftarrow 0$ *to* $v_l - 1$ **do**

```
    begin
    call walk(tree↑.left, n, left labels), appending the results to the buffer;
    call all-walks on the right descendant;
    end
  end
```

4.2 Recognition Problems

We now shift our attention from the generation of strings to the recognition of strings supplied as input. There are two algorithms. The user can supply a string and a grammar and ask the system to verify that the grammar derives the string. This is accomplished with the accepting algorithm. There is also a *ranking* algorithm that carries this process one step further; it returns an integer that uniquely identifies the string within the family generated by the grammar.

4.2.1 Accepting

Accepting a string is a straightforward application of dynamic programming, a technique that many authors have used to parse context free languages. We assume that the input string is of length l, and we construct for every nonterminal an $(l+1) \times (l+1)$ upper triangular matrix with entries initially false. The (i, j) entry will be set true if the nonterminal can derive the piece of string between locations i and $j-1$ inclusive.

There is one other $(l+1) \times (l+1)$ upper triangular matrix M, called the minimums matrix, whose (i, j)th entry contains the smallest label between locations i and $j-1$. The minimums matrix is easily computed with the recurrence $M(i, j) = \min(M(i, j-1), M(i+1, j))$.

The nonterminal matrices are more difficult to compute. Their entries are processed in increasing order of the difference $j - i$. Initially the near diagonal entries $(i, i+1)$ are set true for any nonterminal with a production $P \rightarrow a$, where a matches the input at location i. The upper right corners are computed last. A true entry in the upper right corner of the start nonterminal matrix indicates a successful derivation.

For a given location in the matrices, (i, j), all matrices are computed together, using the safe order among nonterminals described already in Section 4.1.1. When a nonterminal P with production $P \rightarrow QR$ is processed, the production is matched with the input string between i and $j-1$, with the understanding that Q and R can derive variable length strings, so we must try all possible midpoints m in the range $i < m < j-1$. An outline of the algorithm follows.

```
for s ← 2 to length(input) do
  for i ← 1 to length(input) − s + 1 do
    begin
    j ← i + s
    for each nonterminal P in safe order do
      begin
```

```
    P(i,j) ← false
    for each production possibility P → QR do
      for m ← i + 1 to j − 1 do
        if Q(i,m) and R(m,j) and check box operator then
          begin
          P(i,j) ← true
          goto found
          end
    found:
    end
  end
```

The presence of a box operator adds an additional check to the procedure. If a box is on a nonterminal, such as Q, then each time the Q matrix entry (i,m) is tested for true we also check if $M(i,m) = M(i,j)$. Equality insures that, if the production were used to derive the string between i and $j-1$, then the smallest label would appear in the proper location.

The running time of the accepting algorithm is $O(n(l+1)^3)$, where l is the length of the input string, and n is the number of nonterminals. Each of the $O(n(l+1)^2)$ matrix entries treated by the algorithm can require an $O(l)$ string match in the inner loop; together these derive the time bound.

4.2.2 Ranking

The ranking algorithm makes use of the following data structures. The first two are modifications of those used for accepting:

1) Nonterminal substring matrices, $P(i,j)$. Previously these contained true or false, depending on whether the nonterminal could derive the string between i and $j-1$. Now they contain false or the rank of the derived string. We assume that the grammar is unambiguous; ambiguous grammars would require a set of ranks.

2) Minimums matrix, $M(i,j)$. This used to contain the minimum label in the string between i and $j-1$. Now each entry contains an ordered list of all the labels used between i and $j-1$, with the smallest label first.

The third data structure was used before in counting:

3) Nonterminal count matrices, $P_{\mathbf{p}}$. These contain the total number of derivations that start with the nonterminal P, and have a final string with characteristic vector $\mathbf{p}$.

4) Production offset matrices, $(P \rightarrow QR)_{\mathbf{qr}}$. This is a new data structure that will be explained shortly. Each production has a matrix of integers that is addressed by two characteristic vectors, $\mathbf{q}$ and $\mathbf{r}$, corresponding to nonterminals on the right side of the production.

Note that there are two distinctly different matrices associated with each nonterminal. The substring matrix will always be two dimensional (since it is addressed by two locations in the string), and will be denoted with the indices following the nonterminal character: $P(i,j)$. The count matrix has a variable number of dimensions d depending on the number of characters critical to the counting. It is denoted with subscripted indices P_{jkl} or with a vector as a subscript $P_{\mathbf{p}}$. The subscripting distinguishes count matrices from substring matrices.

The accepting algorithm of Section 4.2.1 is used as a basis for the ranking algorithm of this section. Instead of setting $P(i,j)$ true in the inner loop of the code, we actually compute the rank by the formula

$$P(i,j) \leftarrow \Big(s\, Q_{\mathbf{q}} + Q(i,m) \Big)\, R_{\mathbf{r}} + R(m,j) + (P \rightarrow QR)_{\mathbf{qr}}\,. \tag{4.6}$$

where most of the above formula needs further explanation. In essence, the formula inverts the operations of Section 4.1.2 for selection. Whereas equations (4.5) divided the control integer c into three parts by modular arithmetic, Equation (4.6) above assembles three parts, s, $Q(i,m)$, and $R(m,j)$ into one integer.

In Section 4.1.2, s was used to split a list of labels into two groups. Here s is computed by merging two groups of labels: those used in the Q substring (found in $M(i,m)$) and those used in the R substring (found in $M(m,j)$).

$s \leftarrow 0$;
until *a and b lists are empty* **do**
  if $a_{head} < b_{head}$ **then**
    begin
    move the a list head into the output
    $a \leftarrow a - 1$
    end
  else
    begin
    $s \leftarrow s + \binom{a-1}{a}\binom{b}{1}$
    move the b list head into the output
    $b \leftarrow b - 1$
    end

This accounts for all of equation (4.6), save the last, *offset*, term $(P \rightarrow QR)_{\mathbf{qr}}$ whose purpose is to invert the trivial nodes of Section 4.1.2. The trees of trivial nodes between drastic nodes result from a summing process; when trivial nodes are used for selection, a search based on the control integer c guides the program through the trivial nodes to a particular drastic node, and the firing of a specific production. A subrange of the possible values of c will all land at the same drastic node. In the inverse direction, we have computed an integer for the firing of a production and we must add an offset for the base of the subrange. The offset allows for the contribution of other terms in the sum.

Offset matrices can be computed from the walk structure by applying the procedure *compute_offset*$(P_{\mathbf{p}}, 0)$ to each entry of the nonterminal count matrices:

```
procedure compute_offset(tree, sum)
  case tree↑.node_type do
    trivial:
      begin
      call compute_offset(left descendant, sum);
      sum ← sum + value v at current node;
      call compute_offset(right descendant, sum):
      end
    drastic:
      (production)(char. of 1st nonterm)(char. of 2nd nonterm) ← sum;
  end
```

This completes the inversion of the selection algorithm. The running time is the same as the accepting algorithm, $O(n(l+1)^3)$, since the only significant extra computation is the merging of the lists, which is $O(l)$ and so can be absorbed by the $O(l)$ time used to match strings. The major new complication of the ranking algorithm is the $O(m^{2d})$ sized offset matrices; it is ironic that the simplest nodes of Section 4.1.2 are the source of the most complication for the data structures of this section.

4.3 Overall Evaluation

For the reader interested in the details of implementation, there is a PASCAL version of the counting, selection, and generation at random algorithms in Appendix C, along with a sample program execution. In the appendix the strict requirement for Chomsky normal form is relaxed, to allow the user more flexiblity when entering grammars.

While theoretically the running times and space requirements of the algorithms are all polynomial, the implementation experience highlights two limitations likely to cause trouble:

1) The integer size of the host computer. Most families of objects grow exponentially, and so even the simplest counting algorithm must use large integers. Our memory estimates have been based on the number of integers, not on their total number of bits.

2) The memory requirements. This becomes particularly acute when the number of critical characters, d, exceeds two, so that the memory needed for selection or random generation is worse than $O(m^4)$, where m is the maximum number of occurrences of any critical character.

On a 36 bi' PDP-10 computer, with 256K words of virtual memory, and typical grammars from Chap'er 2, these limitations where provoked when $m \approx 12$ and $d \approx 3$.

The algorithms of this chapter constitute a general purpose generation and recognition system for combinatorial objects. For a specific problem, such as the generation of random labelled trees, there are special purpose algorithms that will do better than the general approach. If necessary, trees can be generated in $O(m)$ time rather than $O(m^2)$ time.

Nevertheless the value of a general system is clear: rather than writing a computer program for each new problem, the user is designing a compact grammar. In fact it is easy to describe combinatorial families with grammars of fewer than 8 productions for which no special purpose algorithm exists. (See the sample run in Appendix C for trees counted by leaves and single descendant nodes.)

On the other hand, there are frameworks for selection, enumeration, and generation at random that are more general than the labelled grammars described above. Most notably, Wilf has developed the idea of a path through a directed graph, each node having a variable number of labelled outward arcs [Wil 1977 and 1978] Almost any combinatorial family with a recurrence relation can be placed in one-to-one correspondence with a graph that resembles the recurrence, and then such problems as generation at random and selection are solved with general purpose algorithms on the graph.

The chief distinction of the techniques of this chapter are that, rather than using a single path through a graph, the paths are allowed to fork at certain, *drastic*, nodes. While this branching complicates the internal workings of the algorithms, it permits a greatly simplified specification of the combinatorial family, and essentially automates the one-to-one correspondence between traversals of the graph and combinatorial objects.

In conclusion, it is worth asking a more reflective question: how can the polynomial algorithms of this chapter generate objects from families that are inherently exponential in size? This is possible because of the high degree of decomposability present in families that can be described with labelled formal languages. When a nonterminal N appears in a partially derived string it is treated in a manner that is independent of the size and contents of the string. The nature of the enclosing string may affect the labels given to the string derived from N, but the processing of N uses a arbitrary list of labels that can contain any ordered subset of the integers. This independence, apart from the splitting of labels, makes special treatment for substrings unnecessary--the walk structure can be highly folded, with numerous pointers to small subproblems that are used repeatedly in the construction of larger objects. This is the source of the savings that make it possible to encode an exponentially sized family with a polynomial sized data structure.

CHAPTER 5

CONCLUSION

If we think of enumeration problems as falling into three broad classes: those pertaining to unlabelled structures, those requiring a labelling with distinct labels, and those making repeated use of labels, then the main result of this thesis is to extend the use of formal languages from unlabelled to distinctly labelled problems.

Of course it is natural to ask why we bother with formal languages if we already have combinatorial techniques to deal with these problems. There are several reasons. First, the formal language approach is general. We have seen already in Chapter 2 that many of the classic generating functions can be derived from labelled formal languages. Second, there is an extremely close connection between specification and enumeration. Once we have a formal language description we obtain immediately an integral equation for an enumerating generating function. Formal languages have been extensively used for specification, often in applications where enumeration is not important, so it is natural to tie a large class of enumeration problems to a familiar descriptive tool.

In computer science two major areas of application follow from the connection forged between language theory and enumeration. We can use labelled formal languages as analytic tools in the analysis of algorithms. Chapter 3 developed this idea on two sample problems, the second of which is a new algorithm for tree searching that is interesting in its own right and should see practical use in appropriate applications. Another major use for labelled formal languages is in controlling a general purpose system for counting, generation at random, and enumeration of combinatorial objects. Chapter 4 describes the algorithms necessary for processing labelled formal languages and appendix C includes a running system of this nature. It is no longer necessary to write a different algorithm for each combinatorial problem.

Perhaps the largest open area of research is in the intermediate problems that are neither unlabelled nor distinctly labelled, but make repeated use of a collection of labels. Mathematically these problems are unified by the Polya-Redfield theory of enumeration [Polya 1937] [Redfield 1927], but as yet there is no systematic way to generate the objects of such a family; Burnside's lemma stands between the counting and the construction of family members. The work of R. C. Read is a significant step in this direction [Read 1978]. His approach eliminates many of the redundant family members before they are checked for independence. But there are many fast, special purpose, algorithms (see for example appendix D) that challenge us to pursue this problem further.

The algorithms of Chapter 4 seem ripe for asymptotic improvement of their running times. Kung and Traub have developed methods for the solution of polynomial equations that improve the running time of the counting algorithm of Section 4.1.1 [Kung 1978]; it would be nice to develop similar techniques for the selection and generation algorithms. The accepting problem is probably equivalent to matrix multiplication, and there are likely to be methods that work considerably faster on limited subsets of labelled languages. Both these problems need further exploration, and the second problem suggests yet another open area: the classification of labelled grammars. As yet there is no easy way to characterize those languages requiring more sophisticated partial orders than those denoted with the box operator, and to separate the various uses of the box operator. For example, the grammars of Sections 2.3.1 (alternating permutations), 2.3.7 (Eulerian numbers), and 2.4.1 (left to right maxima and minima) probably all belong in the same class, and yet they use the box operator in a way that makes them clearly beyond the classification techniques of ordinary formal languages.

Appendix A

Bell Polynomials and Lagrange Inversion

It is frequently necessary to obtain the coefficients of a generating function that is the composition of several well known functions. Suppose that $h = f \circ g$, where f, g and h have Taylor expansions like

$$f = \sum_{i \geq 1} f_i \frac{z^i}{i!}. \tag{A.1}$$

To compute h_i in terms of f_i and g_i we use a matrix arrangement of the Faà di Bruno formula introduced by Jabotinski [Jabotinski 1947]. Each function f has a matrix:

$$\begin{pmatrix} m_{11} & m_{12} & m_{13} & \cdots \\ & m_{22} & m_{23} & \cdots \\ & & m_{33} & \cdots \\ & & & \cdots \end{pmatrix}, \tag{A.2}$$

where m_{ij} is an exponential Bell polynomial of the coefficients of f:

$$m_{ij} = B_{ji}(f_1, f_2, \ldots). \tag{A.3}$$

(Note the unfortunate transposition of indices.) These polynomials are defined implicitly by the equation:

$$e^{yf(x)} = \sum_{i,j} B_{ji}(f_1, f_2, \ldots)\, y^i \frac{z^j}{j!}, \tag{A.4}$$

from which we can derive various other expressions for B_{ji}:

$$\begin{aligned} B_{ji} &= \frac{j!}{i!} \langle x^j \rangle \left(f(x)\right)^i \\ B_{ji} &= \sum_{\substack{k_1+k_2+k_3+\cdots=i \\ k_1+2k_2+3k_3+\cdots=j}} \frac{j!}{k_1!\,k_1!\,k_3!\ldots} \left(\frac{f_1}{1!}\right)^{k_1} \left(\frac{f_2}{2!}\right)^{k_2} \left(\frac{f_3}{3!}\right)^{k_3} \cdots. \end{aligned} \tag{A.5}$$

The first few polynomials are

$$\begin{pmatrix} f_1 & f_2 & f_3 & \cdots \\ & f_1^2 & 3f_1f_2 & \cdots \\ & & f_1^3 & \cdots \end{pmatrix}. \tag{A.6}$$

By summing along the columns we obtain another family of polynomials that are associated with Bell's name:

$$Y_j = \sum_{i \geq 1} B_{ji}. \tag{A.7}$$

The top row of the matrix contains the the Taylor expansion of f and each successive row can be computed with the following algorithm:

```
for j ← 1 to ∞ do
  begin
  m_{1j} ← f_j
  for i ← 2 to j do
    begin
    m_{ij} ← 0
    α ← j      (α = (j choose k) in the loop below)
    for k ← 1 to j − i + 1 do
      begin
      m_{ij} ← m_{ij} + α f_k m_{i−1,j−k}
      α ← α (j−k)/(k+1)
      end
    m_{ij} ← m_{ij}/i
    end
  end
```

The most useful property of these matrices is given by the following theorem.

Theorem. *If M_f and M_g are the matrices of f and g then $M_h = M_f M_g$ is the matrix for the composition, $h = f \circ g$.*

Proof:

$$\begin{aligned} e^{yf(g(z))} &= \sum_{i,j} M_f(i,j)\, y^i \frac{g(z)^j}{j!} \\ &= \sum_{i,j} M_f(i,j)\, y^i \left\langle x^j \right\rangle e^{xg(z)} \\ &= \sum_{i,j,k} M_f(i,j) M_g(j,k)\, y^i \frac{z^k}{k!} \\ &= \sum_{i,k} M_h(i,k)\, y^i \frac{z^k}{k!}\,. \end{aligned} \tag{A.8}$$

For the Taylor expansion of h only the top row of M_h need be computed:

Corollary. *If $h = f \circ g$ then*

$$(h_1, h_2, h_3, \ldots) = (f_1, f_2, f_3, \ldots)\, M_g\,. \tag{A.9}$$

Theorem. *If a function f is given implicitly by $h(f) = z$ then the matrix for f is*

$$M_f(i,j) = \frac{(j-1)!}{(i-1)!} \left\langle z^{j-i} \right\rangle \left(\frac{h(x)}{x} \right)^{-j}\,. \tag{A.10}$$

Proof: *[Comtet 1974; p. 149] Consider the product of M_h and M_f, where*

$$M_h(i,j) = \frac{j!}{i!} \langle x^j \rangle h(x)^i \tag{A.11}$$

by equation (A.5).

$$\begin{aligned} M_h(i,j)M_f(j,k) &= \sum_{i \le j \le k} \frac{j!}{i!} \left(\langle x^j \rangle h(x)^i \right) \frac{(k-1)!}{(j-1)!} \left(\langle x^{k-j} \rangle \left(\frac{h(x)}{x} \right)^{-k} \right) \\ &= \frac{(k-1)!}{(i-1)!} \sum_{i \le j \le k} \left(\langle x^{j-1} \rangle h(x)^{i-1} h'(x) \right) \left(\langle x^{k-j} \rangle \left(\frac{h(x)}{x} \right)^{-k} \right) \\ &= \frac{(k-1)!}{(i-1)!} \langle x^{k-1} \rangle x^k h(x)^{i-k-1} h'(x) \\ &= \delta_{ik} \end{aligned} \tag{A.12}$$

Corollary. *(Lagrange inversion) If a function f is given implicitly by $h(f) = z$ then the coefficients in the power series expansion for f are:*

$$f_j = (j-1)! \langle x^{j-1} \rangle \left(\frac{h(x)}{x} \right)^{-j}. \tag{A.13}$$

Here are two frequently encountered matrices:

$$f(x) = e^x - 1$$
$$M_f = \begin{pmatrix} \left\{ {1 \atop 1} \right\} & \left\{ {2 \atop 1} \right\} & \cdots \\ & \left\{ {2 \atop 2} \right\} & \text{Stirling Numbers, 2nd Kind} \\ & & \cdots \end{pmatrix} \tag{A.14}$$

$$g(x) = -\ln(1-x)$$
$$M_g = \begin{pmatrix} \left[{1 \atop 1} \right] & \left[{2 \atop 1} \right] & \cdots \\ & \left[{2 \atop 2} \right] & \text{Stirling Numbers, 1st Kind} \\ & & \cdots \end{pmatrix}$$

As an example of the techniques of this appendix, consider the following equations from Section 2.3.4:

$$\begin{aligned} T &= x e^T \\ P &= e^{-b \ln(1-T)}. \end{aligned} \tag{A.15}$$

The coefficient of b^j in P is the composition of three functions: $\frac{x^j}{j!}$, $-\ln(1-x)$, and $T(x)$. We can compute the matrix for T according to (A.10):

$$\begin{aligned} M_T(k,n) &= \frac{(n-1)!}{(k-1)!} \langle T^{n-k} \rangle e^{nT} \\ &= \binom{n-1}{k-1} n^{n-k}. \end{aligned} \tag{A.16}$$

The matrix for $-\ln(1-x)$ is given above,

$$M_{-\ln(1-x)}(j,k) = \begin{bmatrix} k \\ j \end{bmatrix}, \tag{A.17}$$

and the first row of the matrix for $\frac{x^j}{j!}$ has a 1 in the jth column. Multiplying these together we obtain an expression for the coefficients of P, given already in equation (2.45):

$$\left\langle b^j \frac{x^n}{n!} \right\rangle P = \sum_k \binom{n-1}{k-1} n^{n-k} \begin{bmatrix} k \\ j \end{bmatrix}. \tag{A.18}$$

APPENDIX B

DIMINISHED TREE SEARCH

1. Arbitrary Median Tree Searching. The following algorithm combines a tree search with a linear search. When the tree search portion of the code hits a leaf, it branches to additional code to search a small linear list of size at most $2t$. If insertions cause the linear list to exceed size $2t$ then a new tree node is created and the list is split into two lists of size t. The additional tree balancing provided by the median of $2t + 1$ splitting improves the running time of the algorithm.

```
  define plus_inf ≡ maxint
  define half_size = 3   { This is t. }
  define median = half_size + 1
  define overflow = half_size + half_size + 2
program search(tty, output);
  type link = ↑node; node_type = (tree, list);
    node = record case node_type of
      tree: (left : link; key : integer; right : link);
      list: (keys : array [1 .. overflow − 1] of integer);
      end;
  var p: link; in_key: integer;
    ⟨Declare the Initialize Procedure 6⟩;
    ⟨Declare the Find or Insert Procedure 2⟩;
    begin initialize(p);
    while true do
      begin write(tty, 'Key>␣'); read(tty, in_key); writeln(tty, find(in_key, p));
      end;
    end.
```

2. A Standard Tree Search Algorithm.

The find procedure takes a key, k, and a pointer to a tree, p. If k is in the tree, then find returns *true*, otherwise *false* is returned, and k is inserted into the tree. Most of the work is performed by the inner loop of a standard tree search algorithm. Since the loop is included without modification, any improvements that the programmer might design for standard tree searching (such as hand coding the inner loop) will also be improvements to diminished tree searching.

In order to activate the linear list code without modifying this loop the leaves of the tree contain a dummy $+\infty$ key, a nil left pointer, and a right pointer to a linear list. The standard tree search will find the nil pointer and take the "unsuccessful" loop exit.

```
define success = 1
define quit = 2
```

```
⟨ Declare the Find or Insert Procedure 2 ⟩ ≡
function find(k : integer; p : link): boolean;
  label success, quit;
  var i, temp: integer; left_list, left_tree, right_list, right_tree: link;
  begin while true do
    with p↑ do
      begin if k > key then
        begin p ← right;
        end
      else begin if k = key then goto success
        else if left = nil then ⟨ Search the right linear list, and goto success or quit 3 ⟩
          else p ← left;
        end;
      end;
success: find ← true;
quit: end
```

This code is used in section 1.

3. Searching a Linear List.

The following code is a standard linear list search. It is designed to work for arbitrary list sizes (controlled by the *half_size* parameter). Even though this code is executed only once for each search, as opposed to $O(\log n)$ passes through the loop in the preceding section, the programmer should take care to make this run efficiently; $\log n$ is close to a constant. In particular, the code below is NOT THE BEST approach to the problem since *half_size* is fixed during execution. It is better to choose a fixed *half_size* and unroll the loop, as exhibited in the special median of five variation following this code.

```
⟨ Search the right linear list, and goto success or quit 3 ⟩ ≡
  begin i ← 1;
  with right↑ do
    begin while true do
      begin if keys[i] < k then
        begin i ← i + 1;
        end
      else begin if keys[i] = k then goto success
        else begin ⟨ Insert k and Shift Subsequent Keys 4 ⟩
          find ← false; goto quit;
          end;
        end;
      end;
    end;
  end
```

This code is used in section 2.

4. Inserting a Key into a Linear List.

```
⟨ Insert k and Shift Subsequent Keys 4 ⟩ ≡
  begin repeat temp ← k; k ← keys[i]; keys[i] ← temp; i ← i + 1
  until k = plus_inf;
  if i = overflow then ⟨ Split the Linear List 5 ⟩
  else keys[i] ← plus_inf;
  end
```

This code is used in section 3.

5. Splitting a Linear List.

A linear list has grown to size $2 * half\_size + 1$ so it is split. The median element becomes a new tree node with right and left dummy nodes for sons. The dummy nodes each point to lists of *half_size* keys.

```
⟨ Split the Linear List 5 ⟩ ≡
  with p↑ do
    begin left_list ← right; key ← left_list↑.keys[median];
    left_list↑.keys[median] ← plus_inf; new(right_list, list);
    for i ← 1 to half_size do
      begin right_list↑.keys[i] ← left_list↑.keys[median + i];
      end;
    right_list↑.keys[median] ← plus_inf; new(left, tree); new(right, tree);
    left↑.key ← plus_inf; right↑.key ← plus_inf; left↑.left ← nil; right↑.left ← nil;
    left↑.right ← left_list; right↑.right ← right_list;
    end
```

This code is used in section 4.

6. Initialize the data structure with a tree node and an empty list.

```
⟨ Declare the Initialize Procedure 6 ⟩ ≡
procedure initialize(var p : link);
  begin new(p, tree); p↑.key ← plus_inf; p↑.left ← nil; new(p↑.right, list);
  p↑.right↑.keys[1] ← plus_inf;
  end
```

This code is used in section 1.

1. Median-of-Five Tree Search.

```
  define plus_inf ≡ maxint
  define reps = 5
program search(tty, output);
  type link = ↑node; node_type = (tree, list); status = 'a' .. 'e'; node =
        record case node_type of
      tree: (left : link; key : integer; right : link);
      list: (next_free : status; a, b, c, d : integer);
      end;
  var t: link; in_key: integer;
    heap_bottom, time, test_size, start, test_number, j, test_exponent: integer;
    result: boolean; ⟨Initialize Procedure 2⟩;
    ⟨Find or Insert Procedure 3⟩;
    begin mark(heap_bottom);
    writeln(output, 'Special␣Median␣of␣Five', reps : 6, '␣␣reps␣of␣each␣size');
    test_size ← 1;
    for test_exponent ← 1 to 14 do
      begin test_size ← 2 * test_size; start ← runtime;
      for test_number ← 1 to reps do
        begin initialize(t);
        for j ← 1 to test_size do result ← find(trunc(random(0) * (plus_inf − 1)), t);
        release(heap_bottom);
        end;
      time ← runtime; writeln(output, 'Test␣Size: ', test_size : 5, '␣Time: ',
          (time − start) : 7, '␣Time/Size: ', ((time − start)/test_size) : 8 : 4);
      end;
    end.
```

2. Initialize the data structure with a tree node and an empty list.

```
⟨Initialize Procedure 2⟩ ≡
procedure initialize(var t : link);
  begin new(t, tree);
  with t↑ do
    begin key ← plus_inf; left ← nil; new(right, list); right↑.next_free ← 'a';
    end;
  end
```

This code is used in section 1.

3. A Standard Tree Search Algorithm.

```
  define success = 1
  define quit = 2
  define a_a = 101   { These labels will be used in the block move code }
  define a_b = 102
  define a_c = 103
  define a_d = 104
  define a_e = 105
  define b_b = 106
  define b_c = 107
  define b_d = 108
  define b_e = 109
  define c_c = 110
  define c_d = 111
  define c_e = 112
  define d_d = 113
  define d_e = 114
  define e_e = 115
  define wrap_up = 120
⟨ Find or Insert Procedure 3 ⟩ ≡
function find(k : integer; t : link): boolean;
  label success, quit, a_a, a_b, a_c, a_d, a_e, b_b, b_c, b_d, b_e, c_c, c_d, c_e, d_d, d_e, e_e,
        wrap_up;
  var right_list: link; e: integer;
  begin while true do
    with t↑ do
      begin if k > key then t ← right
      else begin if k = key then goto success
        else if left = nil then ⟨ Linear List Search 4 ⟩
          else begin t ← left;
            end;
        end;
      end;
success: find ← true;
quit: end
```

This code is used in section 1.

4. Searching the Accumulator. Since the search loop is unrolled, a binary search is easy to implement:

```
⟨ Linear List Search 4 ⟩ ≡
  with right↑ do
    begin case next_free of
    'a': goto a_a;
    'b': begin if k < a then goto a_b;
      if k = a then goto success
      else goto b_b
      end;
    'c': if k < b then
        begin if k < a then goto a_c;
        if k = a then goto success
        else goto b_c
        end
      else if k = b then goto success
        else goto c_c;
    'd': if k < b then
        begin if k < a then goto a_d;
        if k = a then goto success
        else goto b_d
        end
      else begin if k < c then
          begin if k = b then goto success
          else goto c_d
          end
        else begin if k = c then goto success
          else goto d_d
          end
        end;
    'e': if k < c then
        begin if k < b then
          begin if k < a then goto a_e;
          if k = a then goto success
          else goto b_e
          end
        else if k = b then goto success
          else goto c_e;
        end
      else begin if k < d then
          begin if k = c then goto success
          else goto d_e
          end
        else begin if k = d then goto success
          else goto e_e
          end
        end;
    end;
```

```
    ⟨Shift Code 5⟩;
    ⟨Split Code 6⟩;
    find ← false; goto quit;
    end
```

This code is used in section 3.

5. Unrolled Block Move.

```
⟨Shift Code 5⟩ ≡
e_e: e ← k; goto wrap_up;
d_e: e ← d;
d_d: d ← k; goto wrap_up;
c_e: e ← d;
c_d: d ← c;
c_c: c ← k; goto wrap_up;
b_e: e ← d;
b_d: d ← c;
b_c: c ← b;
b_b: b ← k; goto wrap_up;
a_e: e ← d;
a_d: d ← c;
a_c: c ← b;
a_b: b ← a;
a_a: a ← k;
```

This code is used in section 4.

6. Wrap up an unsuccessful search, splitting if necessary.

```
⟨Split Code 6⟩ ≡
wrap_up: if next_free ≠ 'e' then next_free ← succ(next_free)
  else begin next_free ← 'c'; new(right_list, list); right_list↑.next_free ← 'c';
    right_list↑.a ← d; right_list↑.b ← e;
    with t↑ do
      begin key ← right↑.c; new(left, tree); left↑.key ← plus_inf; left↑.left ← nil;
      left↑.right ← right; new(right, tree); right↑.key ← plus_inf; right↑.left ← nil;
      right↑.right ← right_list;
      end;
    end;
```

This code is used in section 4.

APPENDIX C

A GENERAL-PURPOSE GENERATOR OF COMBINATORIAL OBJECTS

1. Global Procedures and Parameters.

This program accepts a grammatical description of a family of combinatorial objects, counts the number of specified objects, and generates objects at random or by rank within the family. The grammar must be context free, but it can contain additional "box" operators to control the labelling of one of the terminal characters, so that such things as labelled trees and permutations can be described with grammars. The reader should be familiar with the "box" operator.

There are three major divisions in the code. One section loads a grammar into the program's data structures. Another section counts the number of derivations of the grammar by a process that amounts to tensor multiplication, and at the same time generates a "walk" structure. The third section traverses this walk structure and generates a specified string. The data structures and low level support procedures are shared by all three sections of code.

By convention, the single letter prefix of an identifier, e.g., *x_variable_name*, groups together common variables and types. The following interpretations should help the reader:

i	input
s	symbol table
p	production possibility
m	nonterminal matrices
w	walk structure
l	label list

```
program general(tty);
  type ⟨Global types 8⟩
  var ⟨Global variables 3⟩
    ⟨Global procedures 4⟩
    ⟨Matrix procedures 36⟩
    ⟨Walk structure builders 45⟩
    ⟨Grammar loading procedures 11⟩
    ⟨Counting procedures 49⟩
    ⟨Walking procedures 58⟩
    ⟨Command processing 63⟩
    begin initialize; commands;
    end.
```

2. Various parameters that control the capacity of the program are defined here.

```
define buffer_size = 140  { Maximum number of char on an input line }
define max_prod_symbols = 7  { Symbols in a production possibility }
define max_countable_terminals = 3  { Terminals critical in counting }
define max_total = 12  { Total occurrences of each critical char in the final string }
```

3. The *mult* array will contain the multinomial coefficients, defined by

$$mult[i,j,k] = \frac{(i+j+k)!}{i!\,j!\,k!}$$

```
⟨ Global variables 3 ⟩ ≡
mult: array [−1 .. 1, −1 .. max_total, −1 .. max_total] of integer;
```

See also sections 5, 9, 22, and 43.

This code is used in section 1.

4. Initialization of the multinomial array. The recurrence

$$\binom{i+j+k}{i\ \ j\ \ k} = \binom{i+j+k-1}{i-1\ \ j\ \ k} + \binom{i+j+k-1}{i\ \ j-1\ \ k} + \binom{i+j+k-1}{i\ \ j\ \ k-1}$$

is used to create $mult[i,j,k]$.

```
⟨ Global procedures 4 ⟩ ≡
procedure initialize;
  var i, j, t: integer;
  begin for i ← −1 to max_total do
    begin mult[0, i, −1] ← 0; mult[0, −1, i] ← 0; mult[1, i, −1] ← 0; mult[1, −1, i] ← 0;
    for j ← −1 to max_total do mult[−1, i, j] ← 0;
    end;
  mult[0, 0, 0] ← 1;
  for t ← 1 to max_total do
    for i ← 0 to 1 do
      for j ← 0 to t − i do
        mult[i, j, t−i−j] ← mult[i−1, j, t−i−j]+mult[i, j−1, t−i−j]+mult[i, j, t−i−j−1];
  end;
```

See also sections 6, 7, 23, 24, and 57.

This code is used in section 1.

5. Buffer and variables for input.

```
⟨ Global variables 3 ⟩ +≡
i_buffer: packed array [1 .. buffer_size] of char;
i_scan, i_line_size: integer;  { Current and final positions in the buffer }
help_file: file of char;  { A place to find help }
```

6. Here are two fundamental operations. The macro *capital_letter* is true when its argument is a capital *char* and the function *min* returns its smallest parameter.

```
define capital_letter(#) ≡ ((# ≥ 'A') ∧ (# ≤ 'Z'))
```

```
⟨ Global procedures 4 ⟩ +≡
function min(i, j : integer): integer;
  begin if i < j then min ← i
  else min ← j;
  end;
```

7. The procedure *i_line* reads a line of input into the global buffer, discarding blanks and commas. If *letters_only* is true, the remaining characters must be capital letters. The last character is stored at location *i_line_size* in *i_buffer*.

```
  define start_over = 3
```

```
⟨ Global procedures 4 ⟩ +≡
procedure i_line(letters_only : boolean);
  label start_over;
  var new_scan, original_scan: integer;
  begin start_over: if eoln(tty) then readln(tty);  { A quirk of tty input }
  read(tty, i_buffer : i_line_size); original_scan ← 1; new_scan ← 1;
  while original_scan ≤ i_line_size do
    begin i_buffer[new_scan] ← i_buffer[original_scan];
    if ¬(i_buffer[original_scan] ∈ [' ', ',']) then
      if letters_only ∧ ¬(capital_letter(i_buffer[new_scan])) then
        begin write(tty, 'Error: capital letter(s) expected, try again ...');
        goto start_over;
        end
      else new_scan ← new_scan + 1;
    original_scan ← original_scan + 1;
    end;
  i_line_size ← new_scan − 1; i_scan ← 1;
  end;
```

8. Global Data Structuring.

The central data structure is a symbol table that is indexed by capital letters, and contains information about the letters used in the grammar. An entry for a nonterminal in the symbol table has several pointers. The first (*prods*) points to a list of production possibilities. If, for example, N can be rewritten in three ways, S, T, and U,

$$N \to S \mid T \mid U,$$

then there will be a linked list of three *p_right_side* records hanging from the *prods* field of the N entry in the symbol table. The second (*matrix*) points to a multidimensional array that is indexed by the occurrences of characters critical to the counting. Suppose the user has declared X to be a special labelled character, and F and G to be countable terminal symbols. We can find out the number of strings derived from N that have 6 X's, 5 F's and 4 G's by looking at the (6,5,4) entry in the matrix associated with N in the symbol table. If we were simply interested in counting these occurrences, then an integer entry in this matrix would be adequate. However, we also want to generate objects with the specified numbers of terminal symbols. This is accomplished with a walk structure that will be described later. Each matrix entry is a pointer to the walk structure (*w_ptr*). The integer count of derived objects can be found in the *value* field of the first walk structure node (*w_node*).

The additional fields of a nonterminal in the symbol table (*appears_in*, *derives_empty*, *preceded_by*, and *followers*) are used to derive the safe order of computation. This is an order among nonterminals that will be described in detail later. The *appears_in* field contains a linked list that points to all productions containing the nonterminal; it is a reverse directory for the nonterminals. The boolean *derives_empty* will be set true once it is determined that the nonterminal can derive a pseudo empty string (free of all countable characters). The last step in the computation of the safe order is a topological sort of a partial order among the nonterminals. This partial order is represented by *followers*, a linked list of all nonterminals that must follow in the partial order, and *preceded_by*, an integer counting the occurrences of the nonterminal in other *followers* lists.

⟨ Global types 8 ⟩ ≡

```
a_ptr = ↑a_list; c_ptr = ↑c_list; m_ptr = ↑m_axis; p_ptr = ↑p_right_side;
w_ptr = ↑w_node;
s_type = (undefined, uncount_term, count_term, labelled, nonterm);
s_data = record case status : s_type of
  labelled, count_term: (index : integer);
  nonterm: (prods : p_ptr; matrix : m_ptr; appears_in : a_ptr; derives_empty : boolean;
      preceded_by : integer; followers : c_ptr);
  end;
```

See also sections 10, 21, 35, 39, 44, and 56.

This code is used in section 1.

9. Here is the actual symbol table (*s_table*) and an array containing those terminal symbols critical to the counting (*index_symbols*). The first element of *index_symbols* is always the special labelled character, and *number_indices* records the total number of entries in *index_symbols*. For the example of the preceding section, *index_symbols* would contain X, F and G and *number_indices* would be three. The *s_table* is initialized before each new grammar is input.

```
⟨ Global variables 3 ⟩ +≡
index_symbols: array [1 .. max_countable_terminals] of char;
number_indices: integer;
s_table: array ['A' .. 'Z'] of s_data;
```

10. The production possibility data structure.

Suppose that X is the labelled character, F and G are counted terminals, H is an uncounted terminal, and A and B are nonterminals. Then the production

$$N \rightarrow XFA^{\square}GBGH$$

generates the following record:

1) *string* contains the characters $XFAGBGH$ and *size* is set to the length of *string*, in this case **7**.

2) *sub_string* includes only the nonterminals AB, and *sub_size* is set to **2**.

3) *bxd* is the location of the box operator in *string*. In the above production the third character is boxed.

4) *special_bxd*, *first_bxd*, and *second_bxd* are all either zero or one, depending on whether the special character, the first nonterminal, or the second nonterminal are boxed. Here we have *first_bxd* equal to one, and the others zero.

5) *adjust* contains 1, 1, and 2, corresponding to 1 X, 1 F, and 2 G's in the string. The *adjust* array summarizes the effect of the production on the counts of *index_symbols* in the derived string.

6) *p_left_side* is N.

7) *contributors* is a count of the number of characters in the string that are counted terminals, plus the number of nonterminals that might contribute countable characters. In this example *contributors* is 6.

```
⟨ Global types 8 ⟩ +≡
  characteristic_vector = array [1 .. max_countable_terminals] of integer;
  p_right_side = record size: integer;
    string: array [1 .. max_prod_symbols] of char;
    bxd, special_bxd, first_bxd, second_bxd: integer;
    adjust: characteristic_vector;
    sub_size: integer;
    sub_string: array [1 .. 2] of char;
    p_left_side: char;
    contributors: integer;
    p_next: p_ptr
    end;
```

11. Loading a Grammar.

```
define done_loading_grammar = 1
define flush_production = 2
define done_getting_productions = 3
⟨ Grammar loading procedures 11 ⟩ ≡
procedure loading_grammar;
  label done_loading_grammar, flush_production, done_getting_productions;
  var left_side, cur_char: char; p_possibility: p_ptr; i: integer;
    ⟨ Local variables for computing the safe order 25 ⟩
    ⟨ Local variables for loading matrix entries 54 ⟩
    ⟨ Internal grammar loading procedures 20 ⟩
    ⟨ Safe order procedure 33 ⟩
  begin ⟨ Initialize for computing the safe order 28 ⟩;
  writeln(tty); ⟨ Get declarations 12 ⟩;
  writeln(tty); ⟨ Get productions 14 ⟩;
  writeln(tty); ⟨ Compute the safe order 29 ⟩;
done_loading_grammar: end;
```

This code is used in section 1.

12. The declaration portion of loading a grammar.

The user is asked to classify the letters of the grammar into four categories. A letter is a nonterminal or a terminal that is labelled, counted, or uncounted. Note that the user must supply a single labelled character, even for unlabelled grammars, and that E is a built-in empty string. (The user doesn't need to say E, but E's will not be output in a generated string.)

```
⟨ Get declarations 12 ⟩ ≡
  s_table['E'].status ← uncount_term; write(tty, 'Labelled␣Character>>␣');
  i_line(true);
  while i_line_size ≠ 1 do
    begin write(tty,
        'Enter␣a␣single␣labelled␣character␣(dummy␣if␣necessary)␣...␣');
    i_line(true);
    end;
  index_symbols[1] ← i_buffer[1];
  with s_table[i_buffer[1]] do
    begin ⟨ Reject double definitions by going to done_loading_grammar 13 ⟩;
    status ← labelled; index ← 1;
    end;
  write(tty, 'Counted␣Terminal(s)>>␣'); i_line(true);
  if i_line_size ≥ max_countable_terminals then
    begin writeln(tty, 'Error:␣the␣last␣two␣lines␣have␣more␣than␣',
        max_countable_terminals : 0, '␣characters␣critical␣to␣the␣counting');
    goto done_loading_grammar;
    end;
  while i_scan ≤ i_line_size do
    begin index_symbols[i_scan + 1] ← i_buffer[i_scan];
    with s_table[i_buffer[i_scan]] do
      begin ⟨ Reject double definitions by going to done_loading_grammar 13 ⟩;
      status ← count_term; index ← i_scan + 1;
      end;
    i_scan ← i_scan + 1;
    end;
  number_indices ← i_scan;
  write(tty, 'Uncounted␣Terminal(s)>>␣'); i_line(true);
  while i_scan ≤ i_line_size do
    with s_table[i_buffer[i_scan]] do
      begin ⟨ Reject double definitions by going to done_loading_grammar 13 ⟩;
      status ← uncount_term; i_scan ← i_scan + 1;
      end;
  write(tty, 'Nonterminal(s)>>␣'); i_line(true);
  while i_scan ≤ i_line_size do
    begin with s_table[i_buffer[i_scan]] do
      begin ⟨ Reject double definitions by going to done_loading_grammar 13 ⟩;
      status ← nonterm; matrix ← nil; prods ← nil; appears_in ← nil;
      derives_empty ← false; preceded_by ← 0; followers ← nil;
      end;
    i_scan ← i_scan + 1;
```

end;

This code is used in section 11.

13. Reject redefinitions of a character. The code below checks the previous definition of a character to be sure that it is not being redefined. It always appears in the context of a **with** *s_table*[*i_buffer*[*i_scan*]] **do**.

⟨ Reject double definitions by going to *done_loading_grammar* 13 ⟩ ≡
if *status* ≠ *undefined* **then**
begin *writeln*(*tty*, 'Error:␣', *i_buffer*[*i_scan*], '␣is␣already␣defined');
goto *done_loading_grammar*;
end

This code is used in sections 12, 12, 12, and 12.

14. The production portion of loading a grammar.

Each line of input is scanned. The program expects the left hand side to resemble $N \rightarrow$, followed by a right hand side consisting of production possibilities separated with vertical bars. The box operator appears as an up arrow in the input.

⟨ Get productions 14 ⟩ ≡
begin while *true* **do**
begin *write*(*tty*, 'Production>>␣'); *i_line*(*false*);
if (*i_buffer*[1] = 'U') ∧ (*i_buffer*[2] = 'P') **then goto** *done_getting_productions*;
⟨ Scan the left hand side; **goto** *flush_production* if it has bad format 15 ⟩
⟨ Scan the right hand side; **goto** *done_loading_grammar* if it has bad format 16 ⟩
flush_production: **end**;
done_getting_productions: **end**

This code is used in section 11.

15. The left hand side should consist of a single nonterminal followed by an arrow.

⟨ Scan the left hand side; **goto** *flush_production* if it has bad format 15 ⟩ ≡
left_side ← *i_buffer*[1];
if ¬(*capital_letter*(*left_side*)) **then**
begin *writeln*(*tty*, 'Error:␣production␣should␣begin␣with␣a␣letter');
goto *flush_production*;
end;
if *s_table*[*left_side*].*status* ≠ *nonterm* **then**
begin *writeln*(*tty*, 'Error:␣', *left_side*, '␣should␣be␣a␣nonterminal');
goto *flush_production*;
end;
if *i_buffer*[2] ≠ '→' **then**
begin *writeln*(*tty*, 'Error:␣→␣expected␣after␣', *left_side*); **goto** *flush_production*;
end;

This code is used in section 14.

16. The right hand side may contain several production possibilities separated by vertical bars. The code below rejects small letters and excessively long strings.

```
⟨Scan the right hand side; goto done_loading_grammar if it has bad format 16⟩ ≡
  i_scan ← 3; p_new(p_possibility);
  while i_scan ≤ i_line_size do
    begin case i_buffer[i_scan] of
    '|': ⟨Finish a production possibility 19⟩;
    '^': ⟨Box the previous symbol 17⟩;
    others: begin cur_char ← i_buffer[i_scan];
      if ¬(capital_letter(cur_char)) then
        begin writeln(tty, 'Error: unacceptable character ', cur_char);
        goto done_loading_grammar;
        end;
      with p_possibility↑ do
        begin size ← size + 1;
        if size > max_prod_symbols then
          begin writeln(tty, 'Error: no more than ', max_prod_symbols : 0,
              ' letters in a production possibility');
          goto done_loading_grammar;
          end;
        ⟨Include cur_char in the string 18⟩
        end;
      end
    end; i_scan ← i_scan + 1;
    end;
  ⟨Finish a production possibility 19⟩;
```

This code is used in section 14.

17. An ↑ indicates that the previous symbol is boxed. In the record for the production possibility, *bxd* is set the the location of the boxed character in the string, and one of *first_bxd*, *second_bxd*, or *special_bxd* is set to one, depending on which nonterminal or terminal character is boxed.

```
⟨ Box the previous symbol 17 ⟩ ≡
  with p_possibility↑ do
    begin bxd ← size;
    if size = 0 then
      begin writeln(tty, 'Error: nothing is being boxed'); goto flush_production;
      end;
    case s_table[string[size]].status of
    nonterm: case sub_size of
      1: first_bxd ← 1;
      2: second_bxd ← 1;
      end;
    labelled: special_bxd ← 1;
    others: begin writeln(tty, 'Error: ', string[size], ' cannot be boxed');
      goto done_loading_grammar;
      end
    end;
    end
```

This code is used in section 16.

18. Here is the code that adds a character to the *string* portion of a production possibility. If the character is a nonterminal then it is also added to the *sub_string*. The instructions below appear within the scope of a **with** *p_possibility* **do**.

```
⟨ Include cur_char in the string 18 ⟩ ≡
  string[size] ← cur_char;
  case s_table[cur_char].status of
  labelled, count_term: adjust[s_table[cur_char].index] ← adjust[s_table[cur_char].index]+1;
  nonterm: begin sub_size ← sub_size + 1;
    if sub_size > 2 then
      begin writeln(tty,
          'Error: no more than 2 nonterminals in a production possibility');
      goto done_loading_grammar;
      end;
    sub_string[sub_size] ← cur_char;
    ⟨ Prepare the appearance of a nonterminal for safe order computation 27 ⟩
    end;
  undefined: writeln(tty, 'Error: '. cur_char, ' is undeclared');
  end;
```

This code is used in section 16.

19. Finishing off a production possibility.

The scan has encountered a vertical bar, or the end of the line so it is time to finish a production possibility. There are two cases:
1) The production possibility is free of nonterminals, and so the result of the production can be entered directly in the matrix associated with the left hand side nonterminal at the location of the *adjust* coordinates. (This part of the procedure will be considered later.)
2) The production possibility has nonterminals and so is linked into the *prods* list of the left hand side nonterminal.

```
⟨ Finish a production possibility 19 ⟩ ≡
  begin with p_possibility↑ do
    begin ⟨ Prepare the completion of a production for safe order computation 28 ⟩;
    if adjust[1] > 1 then
      begin writeln(tty, 'Error:␣only␣one␣', index_symbols[1],
          '␣allowed␣in␣a␣production␣possibility'); goto done_loading_grammar;
      end;
    if sub_size = 0 then ⟨ Enter a string of all terminals 55 ⟩
    else begin p_next ← s_table[left_side].prods; s_table[left_side].prods ← p_possibility;
      end;
    write(tty, 'Loaded:␣␣', left_side, '␣→␣');
    for i ← 1 to size do write(tty, string[i], '␣');
    writeln(tty, '␣with␣boxed␣position␣', bxd : 0);
    end;
  p_new(p_possibility);
  end
```

This code is used in sections 16 and 16.

20. Here we initialize a production possibility by creating a new *p_right_side* and setting the appropriate default values.

```
⟨ Internal grammar loading procedures 20 ⟩ ≡
procedure p_new(var p_possibility : p_ptr);
  var i: integer;
  begin new(p_possibility);
  with p_possibility↑ do
    begin bxd ← 0; special_bxd ← 0; first_bxd ← 0; second_bxd ← 0; size ← 0;
    sub_size ← 0;
    for i ← 1 to max_countable_terminals do adjust[i] ← 0;
    end;
  end;
```

This code is used in section 11.

21. Computation of the safe order.
There are two kinds of lists used to compute the safe order. An *a_list* contains pointers to productions, and a *c_list* contains characters representing nonterminals.

```
⟨ Global types 8 ⟩ +≡
  a_list = record prod: p_ptr;
    a_next: a_ptr
    end;
  c_list = record letter: char;
    c_next: c_ptr
    end;
```

22. The *safe_order*, once it is computed, is available throughout the program.

```
⟨ Global variables 3 ⟩ +≡
safe_order, scan_safe_order: c_ptr;
```

23. The *decrement_test_zero* function subtracts one from its operand and returns true when the result is zero.

```
⟨ Global procedures 4 ⟩ +≡
function decrement_test_zero(var operand : integer): boolean;
  begin operand ← operand − 1; decrement_test_zero ← (operand = 0);
  end;
```

24. The *norm* function sums the components of a characteristic vector.

```
⟨ Global procedures 4 ⟩ +≡
function norm(vector : characteristic_vector): integer;
  var index, temp_norm: integer;
  begin temp_norm ← 0;
  for index ← 1 to number_indices do temp_norm ← temp_norm + vector[index];
  norm ← temp_norm;
  end;
```

25. During the safe order computation a depth first search will used to process the the nonterminals. This is implemented with a stack called *unprocessed*.

```
⟨ Local variables for computing the safe order 25 ⟩ ≡
unprocessed, new_unprocessed, new_follower: c_ptr;
new_appearance: a_ptr;
tail_of_safe_order: c_ptr;
scan_s_table, being_processed: char;
```

This code is used in section 11.

26. Initially the stack and safe order are empty.

```
⟨ Initialize for computing the safe order 26 ⟩ ≡
  unprocessed ← nil; safe_order ← nil;
```

This code is used in section 11.

27. The fragment below builds a reverse directory for the nonterminals. It appears in the code that is loading production possibilities. The scan has encountered a nonterminal *cur_char* and already loaded *cur_char* into a *p_possibility*. In addition, a pointer to the *p_possibility* is linked into the *appears_in* list of *cur_char*.

```
⟨ Prepare the appearance of a nonterminal for safe order computation 27 ⟩ ≡
  new(new_appearance);
  with new_appearance↑ do
    begin prod ← p_possibility; a_next ← s_table[cur_char].appears_in;
    end;
  s_table[cur_char].appears_in ← new_appearance;
```

This code is used in section 18.

28. If a production has no characters that are countable, and is free of nonterminals that might derive countable characters, then *contributors* is zero. In this case *left_side* is linked on the *unprocessed* list—it will eventually be processed and marked with *derives_empty* set true.

```
⟨ Prepare the completion of a production for safe order computation 28 ⟩ ≡
  p_left_side ← left_side; contributors ← norm(adjust) + sub_size;
  if contributors = 0 then
    begin new(new_unprocessed);
    with new_unprocessed↑ do
      begin letter ← left_side; c_next ← unprocessed;
      end;
    unprocessed ← new_unprocessed;
    end;
```

This code is used in section 19.

29. The safe order is for computing nonterminal matrices. Suppose there is a production $N \rightarrow A$, where N and A are nonterminals, and we are computing the (i, j, k) entries in all the matrices, then the entry for A must be computed before the entry for N. However, if there are countable terminals, like X and F, added to the right side of the production, $N \rightarrow XAF$, then the computation of N needs an entry from A with a smaller norm. If we compute the entries in the order of ascending norm it is no longer necessary that A precede N. So precedence depends on the presence of critical characters.

Precedence is further complicated by productions like $N \rightarrow AB$, where N, A, and B are all nonterminals. If B might derive a string without countable characters (a pseudo empty string), then A must precede N. For this reason the computation of the safe order is accomplished in two phases. The first phase identifies those nonterminals that derive pseudo empty strings. Then the precedence relations are apparent and the second phase can topologically sort the nonterminals into a linear order that is consistent with these relations.

```
⟨ Compute the safe order 29 ⟩ ≡
  ⟨ Mark those nonterminals deriving pseudo empty strings 30 ⟩;
  write(tty, 'The␣following␣nonterminals␣can␣derive␣pseudo␣empty␣strings:␣');
  for scan_s_table ← 'A' to 'Z' do
    with s_table[scan_s_table] do
      if (status = nonterm) ∧ derives_empty then write(tty, '␣', scan_s_table);
  writeln(tty); ⟨ Count the number of predecessors for each nonterminal 31 ⟩;
  ⟨ Topologically sort the nonterminals into the safe order 32 ⟩;
  write(tty, 'The␣safe␣order␣is:␣'); scan_safe_order ← safe_order;
  while scan_safe_order ≠ nil do
    with scan_safe_order↑ do
      begin write(tty, '␣', letter); scan_safe_order ← c_next;
      end;
  writeln(tty);
```

This code is used in section 11.

30. The nonterminals that can derive pseudo empty strings are identified as follows: when a grammar is loaded any production free of countable terminals and nonterminals causes its left hand side to be placed on the *unprocessed* stack. The loop below removes these "empty derivers" from the stack and, using the *appears_in* lists, finds all occurrences of the empty derivers in other productions and diminishes the *contributors* fields of these productions. When a *contributors* field is reduced to zero another empty deriver has been found, so it is pushed on the *unprocessed* stack.

```
⟨ Mark those nonterminals deriving pseudo empty strings 30 ⟩ ≡
  while unprocessed ≠ nil do
    begin being_processed ← unprocessed↑.letter; unprocessed ← unprocessed↑.c_next;
    with s_table[being_processed] do
      begin derives_empty ← true;
      while appears_in ≠ nil do
        with appears_in↑.prod↑ do
          begin if decrement_test_zero(contributors) then
            begin new(new_unprocessed);
            with new_unprocessed↑ do
              begin letter ← p_left_side; c_next ← unprocessed;
              end;
            unprocessed ← new_unprocessed;
            end;
          appears_in ← appears_in↑.a_next;
          end;
      end;
    end
```

This code is used in section 20.

31. Once the empty derivers are identified we can find all the precedence relations. If A precedes B then one is added to the *preceded_by* field of B and B is linked in the *followers* list of A.

```
define add_scan_to(#) ≡
        begin new(new_follower);
        with new_follower↑ do
          begin letter ← scan_s_table; c_next ← s_table[#].followers;
          end;
        s_table[#].followers ← new_follower; preceded_by ← preceded_by + 1;
        end
```

⟨ Count the number of predecessors for each nonterminal 31 ⟩ ≡

```
for scan_s_table ← 'A' to 'Z' do
  begin with s_table[scan_s_table] do
    if status = nonterm then
      begin p_possibility ← prods;
      while p_possibility ≠ nil do
        with p_possibility↑ do
          begin if norm(adjust) = 0 then
            case sub_size of
            1: add_scan_to(sub_string[1]);
            2: begin if s_table[sub_string[1]].derives_empty then
                 add_scan_to(sub_string[2]);
               if s_table[sub_string[2]].derives_empty then add_scan_to(sub_string[1]);
               end
            end;
          p_possibility ← p_possibility↑.p_next;
          end;
      end;
  end
```

This code is used in section 20.

32. The topological sort starts with a scan of the symbol table, in order to find nonterminals with zero *preceded_by* fields that are ready to be placed in the safe order.

⟨ Topologically sort the nonterminals into the safe order 32 ⟩ ≡

```
for scan_s_table ← 'A' to 'Z' do
  with s_table[scan_s_table] do
    if status = nonterm then
      if preceded_by = 0 then put_in_safe_order(scan_s_table);
⟨ Check that all nonterminals are in the safe order 34 ⟩;
```

This code is used in section 20.

33. When a nonterminal is moved to the safe order the *preceded_by* fields of each of its followers are diminished by one. If any of these fields reach zero more items are added to the safe order.

```
⟨ Safe order procedure 33 ⟩ ≡
procedure put_in_safe_order(now_safe : char);
  var new_safe_order: c_ptr;
  begin new(new_safe_order); new_safe_order↑.letter ← now_safe;
  if safe_order = nil then safe_order ← new_safe_order
  else tail_of_safe_order↑.c_next ← new_safe_order;
  tail_of_safe_order ← new_safe_order; tail_of_safe_order↑.c_next ← nil;
  with s_table[now_safe] do
    begin preceded_by ← -1;
    while followers ≠ nil do
      with followers↑ do
        begin if decrement_test_zero(s_table[letter].preceded_by) then
          put_in_safe_order(letter);
        followers ← c_next;
        end;
    end;
  end;
```

This code is used in section 11.

34.

```
⟨ Check that all nonterminals are in the safe order 34 ⟩ ≡
  for scan_s_table ← 'A' to 'Z' do
    with s_table[scan_s_table] do
      begin if status = nonterm then
        if preceded_by ≠ -1 then
          begin writeln(tty, 'Error: there must be a cycle in the grammar');
          goto done_loading_grammar;
          end;
      end
```

This code is used in section 32.

35. Matrix Data Structure and Procedures.

The data structure and code that follows implements matrices with a variable number of dimensions. There are *number_indices* dimensions all of size *max_total* + 1. Each nonterminal will have one of these matrices containing pointers to the walk structure.

A matrix consists of a series of one dimensional axes. The lower dimensional (interior) axes contain pointers to other axes; only the highest dimensional (exterior) axes hold the contents of the matrix.

Normally access to the matrix requires a pointer dereference for each dimension. To avoid this, the array is addressed with a *m_coord* that has extra fields to retain these pointers. If the same location is accessed, or if a change is made only in the highest dimension then the *m_coord* will still hold the relevant pointers.

An *m_ptr* is a pointer to an *m_axis*.

```
⟨ Global types 8 ⟩ +≡
  m_axis_type = (interior, exterior);
  m_axis = array [0 .. max_total] of
      record case m_axis_type of
      interior: (next_level : m_ptr);
      exterior: (entry : w_ptr)
      end;
  m_coord = record symbol: char;  { This m_coord accesses the matrix for nonterminal
          symbol }
    int: array [1 .. max_countable_terminals] of integer;  { Integer coordinates }
    ptr: array [1 .. max_countable_terminals] of m_ptr  { Pointer coordinates }
    end;
```

36. Allocation and initialization of matrix axes.

```
⟨ Matrix procedures 36 ⟩ ≡
procedure m_new_axis(i : integer; var a : m_ptr);
  var j: integer; w: w_ptr;
  begin new(a);
  if i = number_indices then  { This is the highest dimension. }
    for j ← 0 to max_total do
      begin new(w, trivial);
      with w↑ do
        begin value ← 0; w_next ← nil;
        end;
      a↑[j].entry ← w;
      end
  else  { This is an interior dimension. }
  for j ← 0 to max_total do a↑[j].next_level ← nil;
  end;
```

See also sections 37, 40, and 41.

This code is used in section 1.

37. This is a preliminary access procedure. It allocates unprobed axes if necessary, initializes them, and stores helpful pointers in the *m_coord* record. The *m_locate* procedure must be applied to an *m_coord* before any *m_access* operations.

```
⟨ Matrix procedures 36 ⟩ +≡
procedure m_locate(var m : m_coord);
  var i: integer;
  begin with m do
    with s_table[symbol] do
      begin if matrix = nil then m_new_axis(1, matrix);
      ptr[1] ← matrix;
      for i ← 2 to number_indices do
        begin if ptr[i − 1]↑[int[i − 1]].next_level = nil then
          m_new_axis(i, ptr[i − 1]↑[int[i − 1]].next_level);
        ptr[i] ← ptr[i − 1]↑[int[i − 1]].next_level;
        end;
      end;
  end;
```

38. Accessing the entries of a matrix.

We assume that *m_locate* has been applied, so that *loc* contains the proper pointers. We then need only use *ptr*[*number_indices*] to find the highest dimensional axis, and *int*[*number_indices*] to find the proper entry in this axis.

```
  define m_access(#) ≡ #.ptr[number_indices]↑[#.int[number_indices]].entry
```

39. To keep track of which characteristic vectors have been computed, there is a global matrix of booleans called *done_already*. The next few modules implement this boolean matrix in a manner similar to the preceding modules.

```
⟨ Global types 8 ⟩ +≡
  m_b_ptr = ↑m_b_axis;
  m_b_axis = array [0 .. max_total] of record case m_axis_type of
      interior: (next_level : m_b_ptr);
      exterior: (entry : boolean)
      end;
```

40. Allocation and initialization of the boolean matrix axes.

```
⟨ Matrix procedures 36 ⟩ +≡
procedure m_b_new_axis(i : integer; var a : m_b_ptr);
  var j: integer;
  begin new(a);
  if i = number_indices then  { This is the highest dimension. }
    for j ← 0 to max_total do a↑[j].entry ← false
  else  { This is an interior dimension. }
  for j ← 0 to max_total do a↑[j].next_level ← nil;
  end;
```

41. Preparing to access the boolean matrix.

```
⟨ Matrix procedures 36 ⟩ +≡
procedure m_b_locate(var m : m_coord);
  var i: integer;
  begin with m do
    begin if done_already = nil then
      begin m_b_new_axis(1, done_already); b_ptrs[1] ← done_already;
      end;
    for i ← 2 to number_indices do
      begin if b_ptrs[i − 1]↑[int[i − 1]].next_level = nil then
        m_b_new_axis(i, b_ptrs[i − 1]↑[int[i − 1]].next_level);
      b_ptrs[i] ← b_ptrs[i − 1]↑[int[i − 1]].next_level;
      end;
    end;
  end;
```

42. Accessing the entries of the boolean matrix.

```
define m_b_access(#) ≡ b_ptrs[number_indices]↑[#.int[number_indices]].entry
```

43. The boolean matrix is a global variable. There is also a global array of pointers to access the matrix.

```
⟨ Global variables 3 ⟩ +≡
done_already: m_b_ptr;
b_ptrs: array [1 .. max_countable_terminals] of m_b_ptr;
```

44. Selector Walk Structure and Construction Procedures.

The walk structure is built by *count* using the procedures *w_multiply*, *w_sum*, *w_build* and *w_single_multiply*. It is used by *walk* to quickly generate a string specified by an integer *selector*. The total number of strings derivable from a node of the walk structure is recorded in the value field of the *w_node*. When walking the structure, *selector* should aways be in the range 0 .. *value* − 1. There are two types of nodes, *trivial* and *drastic*. The *trivial* nodes have *left* and *right* sons. They are formed by the summing process, and so the *value* field is the sum of the values of the two children. The *walk* procedure will turn either left or right at a *trivial* node. The *drastic* nodes are the result of the multiplication process. At a *drastic* node the *walk* procedure must (in an intertwined order):

1) Output the terminal symbols in the production used, *p_used*, and
2) Walk the structures for both the nonterminals, *firs_walk* and *seco_walk*.

A *w_ptr* is a pointer to a *w_node*. The *w_next* field is used to link together nodes that have been summed until they are built into a balanced structure.

```
define w_value(#) ≡ #↑.value

⟨Global types 8⟩ +≡
  w_type = (trivial, drastic);
  w_node = record value: integer;
    w_next: w_ptr;
    case state : w_type of
    trivial: (left, right : w_ptr);
    drastic: (split_factor : integer; p_used : p_ptr; firs_specials, seco_specials : integer;
          firs_walk, seco_walk : w_ptr)
    end;
```

45. The multiply procedure

The *w_multiply* procedure creates a *drastic w_node*. The value field is essentially the product of the two value fields of the nonterminals in *p_used*, with the countable symbols divided according to *first_loc* and *second_loc*. This product is modified by *split_factor* because of the exponential nature of the series in the special labelled character. For example, suppose X is the labelled character, and we are multiplying

$$\left(\sum_i f_i \frac{X^i}{i!}\right)\left(\sum_j g_j \frac{X^j}{j!}\right);$$

then a typical term of the product will be

$$\binom{i+j}{i} f_i g_j \frac{X^{i+j}}{(i+j)!}.$$

The *split_factor* plays the role of the binomial coefficient, although below we are using the multinomial coefficients because there are really three contributions to the product: the number of X's in the production itself, *adjust*[1]; the number of X's in the first nonterminal, *firs_specials*; and the number of X's in the second nonterminal, *seco_specials*. When the *walk* procedure traverses this *w_node* it will have a list of labels that it intends to assign to terminal X's. The *split_factor* counts the number of ways of dividing the list into three parts. If any of the three parts are "boxed" then 1 is subtracted from the corresponding index. Algebraically, this is a consequence of the shifting of the exponential series with the differentiation and integration operators. Combinatorially, this corresponds to fixing the location of the smallest label in the separation process.

```
⟨ Walk structure builders 45 ⟩ ≡
procedure w_multiply(var product : w_ptr; p_possibility : p_ptr;
        first_loc, second_loc : m_coord);
  begin with product↑, p_possibility↑ do
    begin p_used ← p_possibility; firs_specials ← first_loc.int[1];
    seco_specials ← second_loc.int[1]; split_factor ← mult[adjust[1] − special_bxd,
        firs_specials − first_bxd, seco_specials − second_bxd];
    firs_walk ← m_access(first_loc); seco_walk ← m_access(second_loc);
    value ← split_factor * w_value(firs_walk) * w_value(seco_walk); w_next ← nil;
    end;
  end;
```

See also sections 46, 47, and 48.

This code is used in section 1.

46. The multiply procedure with a single operand.

The *w_single_multiply* routine is identical to *w_multiply* except that the production has only one nonterminal.

```
⟨ Walk structure builders 45 ⟩ +≡
procedure w_single_multiply(var product : w_ptr; p_possibility : p_ptr; first_loc : m_coord);
  begin with product↑, p_possibility↑ do
    begin p_used ← p_possibility; firs_specials ← first_loc.int[1];
    split_factor ← mult[adjust[1] − special_bxd, firs_specials − first_bxd, 0];
    firs_walk ← m_access(first_loc); value ← split_factor * w_value(firs_walk);
    seco_specials ← 0; w_next ← nil;
    end;
  end;
```

47. The sum procedure.

The eventual purpose of the *w_sum* procedure is to produce a *w_node* with two descendents, *left* and *right*, and a *value* field equal to the sum of the values of the two descendents. However, in order to optimize the data structure for later walks, the *w_nodes* are first linked together by their *w_next* pointers in an *accumulator* list. Later on, the *w_build* procedure will create a tree.

```
⟨ Walk structure builders 45 ⟩ +≡
procedure w_sum(var accumulator : w_ptr; new : w_ptr);
  begin new↑.w_next ← accumulator; accumulator ← new;
  end;
```

48. A balanced tree builder.

This procedure builds a balanced tree from a list by repeatedly combining the first two nodes and reinserting them at the end of the list.

```
⟨ Walk structure builders 45 ⟩ +≡
procedure w_build(var accumulator : w_ptr);
  var tail, joined: w_ptr;
  begin tail ← accumulator;
  while (tail↑.w_next ≠ nil) do tail ← tail↑.w_next;  { Find the end of the list }
  while accumulator↑.w_next ≠ nil do
    begin new(joined, trivial); joined↑.left ← accumulator;
    joined↑.right ← accumulator↑.w_next;
    joined↑.value ← accumulator↑.value + accumulator↑.w_next↑.value;
    tail↑.w_next ← joined; tail ← joined; tail↑.w_next ← nil;
    accumulator ← accumulator↑.w_next↑.w_next;
    end;
  end;
```

49. Counting Procedures.

The procedure *count* is called with a *loc* parameter that contains a nonterminal and a characteristic vector, indicating that the user desires an accounting of the derivations that start with the nonterminal and have a final composition equal to the vector.

If *loc* is not already computed, the code below builds up a solution to *loc* by first computing all vectors with norm less than *loc*. In the loop below *total* ascends through norm sizes. For a fixed value of *total* the inner loop distributes *total* among the components of the characteristic vector *target_loc*, subject to the constraint that no component can exceed its corresponding component in *loc*. Thus we generate all *target_loc*'s that are componentwise less than *loc*, in ascending order of their norms.

```
define next_total = 10
define next_production = 11
⟨ Counting procedures 49 ⟩ ≡
function count(loc : m_coord): integer;
  label next_total, next_production;
  var partial_sums: array [0 .. max_countable_terminals] of integer;
    chunk, i, col_1, col_2, total, sub_total: integer;
    target_loc, delta_loc, first_loc, second_loc: m_coord;
    scan_productions: p_ptr;
    accumulator, new_drastic: w_ptr;
  begin m_b_locate(loc); { Check done_already to see if loc needs computing }
  if ¬m_b_access(loc) then
    begin new(new_drastic, drastic); new_drastic↑.w_next ← nil;
    sub_total ← 0; partial_sums[0] ← 0;
    for col_1 ← 1 to number_indices do
      begin sub_total ← sub_total + loc.int[col_1]; partial_sums[col_1] ← sub_total;
      target_loc.int[col_1] ← 0;
      end;
    for total ← 0 to partial_sums[number_indices] do
      begin sub_total ← total; col_1 ← number_indices;
      while true do
        begin while sub_total > 0 do { Disperse sub_total leftwards }
          begin chunk ← min(sub_total, loc.int[col_1]); target_loc.int[col_1] ← chunk;
          sub_total ← sub_total − chunk; col_1 ← col_1 − 1;
          end;
        ⟨ Compute all entries at target_loc 50 ⟩
        if sub_total = total then goto next_total;
        while (target_loc.int[col_1 + 1] = 0) ∨ (sub_total = partial_sums[col_1]) do
              { Scan rightwards to find a column to diminish }
          begin col_1 ← col_1 + 1; sub_total ← sub_total + target_loc.int[col_1];
          target_loc.int[col_1] ← 0;
          if sub_total = total then goto next_total;
          end;
        target_loc.int[col_1 + 1] ← target_loc.int[col_1 + 1] − 1; sub_total ← sub_total + 1;
        end;
    next_total: end;
    end;
```

```
  count ← w_value(m_access(loc));
  end;
```

This code is used in section 1.

50. For a particular *target_loc*, the code below scans the nonterminals in *safe_order*, and for each nonterminal it scans the production possibilities.

```
⟨ Compute all entries at target_loc 50 ⟩ ≡
  m_b_locate(target_loc); { Check done_already to see if target_loc needs computing }
  if ¬m_b_access(target_loc) then
    begin m_b_access(target_loc) ← true; scan_safe_order ← safe_order;
    while scan_safe_order ≠ nil do
      with scan_safe_order↑ do
        begin target_loc.symbol ← letter; m_locate(target_loc);
        accumulator ← m_access(target_loc); scan_productions ← s_table[letter].prods;
        while scan_productions ≠ nil do
          with scan_productions↑ do
            begin ⟨ Compute the effect of a particular production 51 ⟩;
            scan_productions ← p_next;
            end;
        w_build(accumulator); m_access(target_loc) ← accumulator;
        scan_safe_order ← c_next;
        end;
    end;
```

This code is used in section 49.

51. At this point we are ready to compute the contribution of a single production. Suppose the production is

$$N \to XFA^{\square}GBGH$$

and we are counting derivations that start with N and have final composition $(6,5,2)$. The code below subtracts the adjust vector contained in the production possibility, $(1,1,2)$, obtaining a *delta_loc* vector of $(5,4,2)$. If the *p_possibility* has only one nonterminal, then all of $(5,4,2)$ must be contributed by this nonterminal. Here, however, there are two nonterminals, so $(5,4,2)$ is divided in all possible ways, as described in the next section.

```
⟨ Compute the effect of a particular production 51 ⟩ ≡
  with target_loc do
    begin { Subtract the counts of characters for this scan_productions from the totals }
    for i ← 1 to number_indices do
      begin delta_loc.int[i] ← int[i] − adjust[i];
      if delta_loc.int[i] < 0 then
        goto next_production { There is no way to use this possibility }
      end;
    delta_loc.symbol ← sub_string[1]; m_locate(delta_loc);
    case sub_size of { 0: p_possibilities without nonterminals are entered directly into
          the matrix }
    1: begin w_single_multiply(new_drastic, scan_productions, delta_loc);
      ⟨ If new_drastic is nonzero then add it to the accumulator 53 ⟩;
      end;
    2: ⟨ Compute with two nonterminals 52 ⟩;
    end;
  next_production: end;
```

This code is used in section 50.

52. Computing the contribution of a production possibility with two nonterminals.

The example of the preceding section has two nonterminals, A and B, in the production possibility that together must contribute $(5,4,2)$ characters to the derived string. The code below divides *delta_loc* (which is $(5,4,2)$ in this example) into *first_loc* and *second_loc* by a counting mechanism. For each partition, the number of strings derived from A is multiplied by the number of strings derived from B, and the results of all partitions are summed together. Multiplication and summation are performed by *w_multiply* and *w_sum*, which are closely related to ordinary multiplication and addition, but take into account the exponential nature of the generating function and the construction of the walk structure.

The code below is the inner loop of the program.

```
⟨ Compute with two nonterminals 52 ⟩ ≡
  begin first_loc ← delta_loc; second_loc.symbol ← sub_string[2];
  for i ← 1 to number_indices do second_loc.int[i] ← 0;
  m_locate(second_loc); w_multiply(new_drastic, scan_productions, first_loc, second_loc);
  ⟨ If new_drastic is nonzero then add it to the accumulator 53 ⟩;
  col_2 ← number_indices;
  while col_2 > 0 do
    begin first_loc.int[col_2] ← first_loc.int[col_2] − 1;
    if first_loc.int[col_2] < 0 then
      begin first_loc.int[col_2] ← delta_loc.int[col_2]; second_loc.int[col_2] ← 0;
      col_2 ← col_2 − 1;
      end
    else begin second_loc.int[col_2] ← second_loc.int[col_2] + 1;
      if col_2 < number_indices then
        begin m_locate(first_loc); m_locate(second_loc);
        end;
      w_multiply(new_drastic, scan_productions, first_loc, second_loc);
      ⟨ If new_drastic is nonzero then add it to the accumulator 53 ⟩;
      col_2 ← number_indices;
      end;
    end;
  end
```

This code is used in section 51.

53. Relevant *new_drastic* nodes are added to the walk structure.

```
⟨ If new_drastic is nonzero then add it to the accumulator 53 ⟩ ≡
  if new_drastic↑.value ≠ 0 then
    begin w_sum(accumulator, new_drastic); new(new_drastic, drastic);
    new_drastic↑.w_next ← nil;
    end
```

This code is used in sections 51, 52, and 52.

54. Addendum to the grammar loading code. Here we define variables for the next module.

```
⟨ Local variables for loading matrix entries 54 ⟩ ≡
loc: m_coord;
new_drastic, already: w_ptr;
```

This code is used in section 11.

55. When a production is free of nonterminals, it is entered directly into the entry of the matrix for the nonterminal on the left side of the production. The code below fits in the context of a **with** *p_possibility* **do**. The *adjust* array of the *p_possibility* describes the makeup of the production; so it is used to address the matrix.

```
⟨ Enter a string of all terminals 55 ⟩ ≡
  with loc do
    begin symbol ← left_side;
    for i ← 1 to number_indices do int[i] ← adjust[i];
    m_locate(loc); new(new_drastic, drastic);
    with new_drastic↑ do
      begin value ← 1; w_next ← nil; split_factor ← 1; p_used ← p_possibility;
      firs_specials ← 0; seco_specials ← 0;
      end;
    already ← m_access(loc); w_sum(already, new_drastic); m_access(loc) ← already;
    end
```

This code is used in section 19.

56. Label Lists.

Now we have completed the procedures that build the data structures, and we are ready to use them. The top level call to the *walk* procedure begins with a list of the integers $1 \,..\, n$. Subsequent calls to *walk* will have fragments of this list, the fragmentation being performed by the *split* procedure. Eventually each integer in the list will become a label for one of the special characters.

```
⟨ Global types 8 ⟩ +≡
  l_ptr = ↑l_list; l_list = record lab: integer;
    l_next: l_ptr;
    end;
```

57. Append a label to the end of the list, using the pointer *l_end* to quickly find the last item of the list.

```
⟨ Global procedures 4 ⟩ +≡
procedure l_append(var l_begin, l_end, new : l_ptr);
  begin if l_begin = nil then l_begin ← new
  else l_end↑.l_next ← new;
  new↑.l_next ← nil; l_end ← new;
  end;
```

58. Procedures to Walk and Produce a String.

According to the value of *selector*, the procedure *split* separates a list of labels, *i*, into three lists, *o1*, *o2*, and *o3* with sizes *n1*, *n2*, and *n3*. The multinomial coefficients govern the splitting process.

```
⟨ Walking procedures 58 ⟩ ≡
procedure split(i : l_ptr; selector, n1, n2, n3 : integer; var o1, o2, o3 : l_ptr);
  var cur, o1_end, o2_end, o3_end: l_ptr;
  begin o1 ← nil; o2 ← nil; o3 ← nil;
  while i ≠ nil do
    begin cur ← i; i ← i↑.l_next;
    if selector < mult[n1 − 1, n2, n3] then
      begin l_append(o1, o1_end, cur); n1 ← n1 − 1;
      end
    else begin selector ← selector − mult[n1 − 1, n2, n3];
      if selector < mult[n1, n2 − 1, n3] then
        begin l_append(o2, o2_end, cur); n2 ← n2 − 1;
        end
      else begin selector ← selector − mult[n1, n2 − 1, n3];
        if selector < mult[n1, n2, n3 − 1] then
          begin l_append(o3, o3_end, cur); n3 ← n3 − 1;
          end
        else write(tty, 'Error: selector inappropriate in split procedure.');
        end;
      end;
    end;
  end;
```

See also section 59.

This code is used in section 1.

59. The actual walking procedure.

The *walk* procedure traverses a data structure that is prepared by other portions of the program (*count*, *w_sum*, *w_multiply*, ...). The purpose of *walk* is to generate a string from the grammar; *selector* specifies the string to be generated, and *labels* is a list of integers that are to be attached to the occurrences of one particular terminal symbol.

The *walk* procedure first dispenses with the trivial nodes by branching left, except when *selector* is larger than the value of the left son, in which case the left value is subtracted from *selector* and the right branch is taken.

Eventually a *drastic* node is reached – these are processed in the next module.

```
⟨ Walking procedures 58 ⟩ +≡
procedure walk(cur : w_ptr; selector : integer; labels : l_ptr);
  var share, so_far, i: integer; the_box, spec_label, firs_labels, seco_labels: l_ptr;
  begin while cur↑.state = trivial do
    if selector < cur↑.left↑.value then cur ← cur↑.left
    else begin selector ← selector − cur↑.left↑.value; cur ← cur↑.right;
      end;
  with cur↑, cur↑.p_used↑ do
    if value > 0 then ⟨ Traverse a drastic node 60 ⟩
  end;
```

60. A *drastic* node indicates that a production is to be applied, so the procedure must divide the labels between the special character and the nonterminals in the production. Part of *selector* is removed (by mixed-radix arithmetic) and used to govern the division process. If anything is boxed then the smallest label is stripped from the beginning of the list, saved in *the_box*, and then returned to the appropriate list after the *split* procedure divides the labels.

```
⟨ Traverse a drastic node 60 ⟩ ≡
  begin if bxd > 0 then {Remove the smallest label}
    begin the_box ← labels; labels ← labels↑.l_next;
    end;
  split(labels, selector mod split_factor, adjust[1] − special_bxd, firs_specials − first_bxd,
      seco_specials − second_bxd, spec_label, firs_labels, seco_labels);
  selector ← selector div split_factor;
  ⟨ Put the smallest label into the appropriate list 61 ⟩;
  ⟨ Output the selected string 62 ⟩;
  end;
```

This code is used in section 59.

61. The smallest label, *the_box*, is linked at the begining of the list corresponding to the box operator in the production.

```
⟨ Put the smallest label into the appropriate list 61 ⟩ ≡
  if special_bxd > 0 then
    begin the_box↑.l_next ← nil; spec_label ← the_box;
    end;
  if first_bxd > 0 then
    begin the_box↑.l_next ← firs_labels; firs_labels ← the_box;
    end;
  if second_bxd > 0 then
    begin the_box↑.l_next ← seco_labels; seco_labels ← the_box;
    end;
```

This code is used in section 60.

62. With the labels ready, the code below scans the production, outputting terminals and calling itself recursively for the nonterminals. The original *selector* is divided into three parts, one part to control the *split* procedure (as we saw above) and two parts to give to the recursive calls on the nonterminals in the production.

```
⟨ Output the selected string 62 ⟩ ≡
  so_far ← 0;
  for i ← 1 to size do
    case s_table[string[i]].status of
    uncount_term, count_term: if string[i] ≠ 'E' then write(tty, string[i], '␣');
    labelled: begin write(tty, string[i], spec_label↑.lab : 0, '␣');
      end;
    nonterm: case so_far of
      0: begin share ← w_value(firs_walk);
        walk(firs_walk, selector mod share, firs_labels); so_far ← 1;
        end;
      1: walk(seco_walk, selector div share, seco_labels);
      end;
    end;
```

This code is used in section 60.

63. Interacting with the user. The help procedure prints information for the user by reading from the file *help.txt* and printing the contents on the *tty*. Parameters that are easy to change are described at the end of the help message.

```
⟨ Command processing 63 ⟩ ≡
procedure help;
  begin reset(help_file, 'HELP.TXT[1,DHG]');
  repeat readln(help_file, i_buffer : i_line_size); writeln(tty, i_buffer : i_line_size);
  until eof(help_file);
  writeln(tty, buffer_size : 0, ' characters on a line of input');
  writeln(tty, max_prod_symbols : 0, ' symbols in a production possibility');
  writeln(tty, max_countable_terminals : 0,
      ' symbols that are critical to the counting');
  writeln(tty, max_total : 0,
      ' total occurrences of each critical symbol in the derived string');
  end;
```

See also section 64.

This code is used in section 1.

64. Command Line Processor.

This procedure consists of three nested loops at three different levels of the command line processor. The outside loop prompts with >, and the label *new_command* marks the end of this loop. The inner loops appear in subsequent modules.

With a few exceptions, the command line processor ignores all but the first letter of a line of input.

```
define quit = 5
define new_command = 6
define new_size = 7
define new_selection = 8
```

```
⟨ Command processing 63 ⟩ +≡
procedure commands;
  label quit, new_size, new_command, new_selection;
  var i, column, selector: integer; c: char; loc: m_coord;
    limit, start_time, heap_bottom: integer; labels, new_label: l_ptr;
  begin writeln(tty);
  writeln(tty, 'Please␣use␣capitals.␣The␣HELP␣command␣prints␣instructions.');
  mark(heap_bottom);
  while true do
    begin writeln(tty); write(tty, 'Command>␣'); i_line(true);
    case i_buffer[1] of
    'H': help; { Help command }
    'G': { Grammar command }
      begin for c ← 'A' to 'Z' do s_table[c].status ← undefined;
      release(heap_bottom); loading_grammar; done_already ← nil;
      end;
    'S': ⟨ Process a size command 65 ⟩;
    'U': goto quit; { Up command }
    others: writeln(tty, 'Error:␣unrecognized␣command')
    end;
  new_command: end;
quit: end;
```

65. Here is the middle loop, it prompts for various structure sizes.

```
⟨ Process a size command 65 ⟩ ≡
  while true do
    begin ⟨ Prompt for the start symbol and the characteristic vector desired 66 ⟩;
    start_time ← runtime; { The user's request is in loc }
    m_locate(loc); writeln(tty, 'There␣are␣', count(loc) : 0, '␣structures');
    writeln(tty, 'Runtime:␣', (runtime − start_time) : 0, '␣msec␣of␣CPU');
    if count(loc) > 0 then ⟨ Walk through selected strings until the user types UP 67 ⟩;
  new_size: end
```

This code is used in section 64.

66. The user specifies a starting symbol for the derivation, and gives the desired distribution of characters for the terminal string.

```
⟨ Prompt for the start symbol and the characteristic vector desired 66 ⟩ ≡
  writeln(tty); write(tty, 'Start␣Nonterminal>>␣'); i_line(true);
  if (i_buffer[1] = 'U') ∧ (i_buffer[2] = 'P') then goto new_command;
  if s_table[i_buffer[1]].status ≠ nonterm then
    begin writeln(tty, 'Error:␣', i_buffer[1], '␣should␣be␣a␣nonterminal');
    goto new_size;
    end;
  loc.symbol ← i_buffer[1];
  writeln(tty, 'Number␣of␣occurrences␣of␣(Limit␣=␣', max_total : 2, ')');
  for i ← 1 to number_indices do
    begin write(tty, index_symbols[i], '''s>>␣'); read(tty, loc.int[i]);
    while (loc.int[i] < 0) ∨ (loc.int[i] > max_total) do
      begin write(tty, 'Error:␣limit␣=␣', max_total : 0, '.␣try␣again␣...␣');
      read(tty, loc.int[i]);
      end;
    end
```

This code is used in section 65.

67. Here is the innermost loop: the user supplies an integer identifying a particular structure, or makes a RANDOM request.

```
⟨ Walk through selected strings until the user types UP 67 ⟩ ≡
  begin while true do
    begin labels ← nil; limit ← loc.int[1];
    for i ← 1 to limit do
      begin new(new_label); new_label↑.lab ← limit − i + 1; new_label↑.l_next ← labels;
      labels ← new_label;
      end;
    writeln(tty); writeln(tty, 'Which␣one␣would␣you␣like?');
    write(tty, 'Enter␣[0␣..␣n-1]␣or␣RANDOM>>␣'); i_line(false);
    case i_buffer[1] of
    'U': goto new_size; { Up command }
    'R': { Random command }
      begin walk(m_access(loc), trunc(random(0) * w_value(m_access(loc))), labels);
      end;
    others: ⟨ Select a particular structure 68 ⟩
    end;
  new_selection: end;
  end
```

This code is used in section 65.

68. The user has requested a specific structure by number.

```
⟨ Select a particular structure 68 ⟩ ≡
  begin  { Convert the contents of i_buffer to a number }
  column ← 1; selector ← 0;
  while column ≤ i_line_size do
    begin if ¬(i_buffer[column] ∈ ['0' .. '9']) then
      begin writeln(tty, 'Error: numeral or command expected');
      goto new_selection;
      end;
    selector ← selector * 10 + ord(i_buffer[column]) − ord('0'); column ← column + 1;
    end;
  if (selector ≥ 0) ∧ (selector < count(loc)) then walk(m_access(loc), selector, labels)
  else writeln(tty, 'Error: selection out of range');
  end
```

This code is used in section 67.

Index

Sample Execution

The preceding program executes as follows:

```
Please use capitals. The HELP command prints instructions.

Command> HELP
Level in the command line processor is indicated by the number of > signs
following a prompt. At any level, the UP command shifts the system to a
higher level. A summary of top level commands follows:
>HELP           Print this file
>UP             Exit from the program
>GRAMMAR        Accept a grammar from the user
>SIZE           Accept a specification from user for the size of objects to
                be counted, selected, or generated at random.

After a GRAMMAR command, the program asks for a classification of input
letters. Letters can be designated as:  1) The special labelled character.
2) Terminals that are not labelled, but figure in the specification of the
problem size.  3) Terminals that will be ignored in the counting, but
printed in the final result.  4) Nonterminals.  At each prompt the user
supplies one or more letters, separated, if desired, by spaces or commas.
E is the built in empty string.  It should not be redefined.

Following the declaration of letters, the program will ask for
productions. It expects a vertical bar to separate production
possibilities, and an up arrow to indicate a box superscript. The UP
command, when issued at the beginning of a line of input, returns the
program to its top level.

In response to a SIZE command, the program asks the user for a start
nonterminal and then the number of occurrences of the special character and
each of the terminals declared to be important to the specification of
problem size. It reports the results of the counting and then asks the
user to select a particular object. The user can supply an integer or use
the RANDOM command for less predictable results.

Here is a sample program execution for labelled trees:
-------------------------------------------------------------------------

Please use capitals. The HELP command prints instructions.

Command> GRAMMAR

Labelled Character>> X
Counted Terminal(s)>>
Uncounted Terminal(s)>> L, H
Nonterminal(s)>> ST
```

```
Production>> T→XS
Loaded:  T → X S  with boxed position 0
Production>> S → L T^ H S | E
Loaded:  S → L T H S  with boxed position 2
Loaded:  S → E  with boxed position 0
Production>> UP

The following nonterminals can derive pseudo empty strings:  S
The safe order is:  T S

Command> SIZE

Start Nonterminal>> T
Number of occurrences of (Limit = 12)
X's>> 3
There are 9 structures
Runtime: 3 msec of CPU

Which one would you like?
Enter [0 .. n-1] or RANDOM>>> 0
X1 L X2 H L X3 H
Which one would you like?
Enter [0 .. n-1] or RANDOM>>> 1
X2 L X1 H L X3 H
Which one would you like?
Enter [0 .. n-1] or RANDOM>>> RANDOM
X2 L X1 H L X3 H
Which one would you like?
Enter [0 .. n-1] or RANDOM>>> UP

Start Nonterminal>> T
Number of occurrences of (Limit = 12)
X's>> 10
There are 1000000000 structures
Runtime: 12 msec of CPU

Which one would you like?
Enter [0 .. n-1] or RANDOM>>> R
X1 L X3 L X9 L X8 L X4 L X7 L X5 L X10 L X2 H H H H H L X6 H H H H
Which one would you like?
Enter [0 .. n-1] or RANDOM>>> UP

Start Nonterminal>> UP

Command> UP

Exit
```

```
^C
.
```

```
The following limits are built into the program:
1 special labelled character in a production possibility.
2 nonterminals in a production possibility.

The following limits can be modified by recompilation:
140 characters on a line of input
7 symbols in a production possibility
3 symbols that are critical to the counting
12 total occurrences of each critical symbol in the derived string
```

The next example uses a grammar to encode Fibonacci sequences. These are strings of + (PS) and − (MS) signs having no consecutive minus signs.

```
Command> GRAMMAR

Labelled Character>>
Enter a single labelled character (dummy if necessary) ... Z
Counted Terminal(s)>> S
Uncounted Terminal(s)>> P M
Nonterminal(s)>> A B

Production>> A → PS A | MS B | E
Loaded:  A → P S A  with boxed position 0
Loaded:  A → M S B  with boxed position 0
Loaded:  A → E  with boxed position 0
Production>> B → PS A | E
Loaded:  B → P S A  with boxed position 0
Loaded:  B → E  with boxed position 0
Production>> UP

The following nonterminals can derive pseudo empty strings:  A B
The safe order is:  A B

Command> SIZE

Start Nonterminal>> A
Number of occurrences of (Limit = 12)
Z's>> 0
S's>> 2
There are 3 structures
Runtime: 3 msec of CPU

Which one would you like?
Enter [0 .. n-1] or RANDOM>>> 0
P S P S
Which one would you like?
Enter [0 .. n-1] or RANDOM>>> 1
P S M S
Which one would you like?
Enter [0 .. n-1] or RANDOM>>> 2
M S P S
Which one would you like?
Enter [0 .. n-1] or RANDOM>>> UP

Start Nonterminal>> A
Number of occurrences of (Limit = 12)
Z's>> 0
S's>> 3
```

```
There are 5 structures
Runtime: 1 msec of CPU

Which one would you like?
Enter [0 .. n-1] or RANDOM>>> UP

Start Nonterminal>> A
Number of occurrences of (Limit = 12)
Z's>> 0
S's>> 4
There are 8 structures
Runtime: 2 msec of CPU

Which one would you like?
Enter [0 .. n-1] or RANDOM>>> UP

Start Nonterminal>> UP
```

This last grammar counts unordered, labelled trees according to their leaves and single descendant nodes. Leaves are marked with an A, and single descendents are marked with a B.

```
Command> GRAMMAR

Labelled Character>> X
Counted Terminal(s)>> A B
Uncounted Terminal(s)>> L H
Nonterminal(s)>> S T M

Production>> T → X S
Loaded:  T → X S  with boxed position 0
Production>> S → A | B L T H | L T^ H M
Loaded:  S → A  with boxed position 0
Loaded:  S → B L T H  with boxed position 0
Loaded:  S → L T H M  with boxed position 2
Production>> M → L T H | L T^ H M
Loaded:  M → L T H  with boxed position 0
Loaded:  M → L T H M  with boxed position 2
Production>> UP

The following nonterminals can derive pseudo empty strings:
The safe order is:  S T M

Command> SIZE

Start Nonterminal>> 3
Error: capital letter(s) expected, try again ... T
Number of occurrences of (Limit = 12)
X's>> 3
A's>> 2
B's>> 0
There are 3 structures
Runtime: 56 msec of CPU

Which one would you like?
Enter [0 .. n-1] or RANDOM>>> 0
X1 L X2 A H L X3 A H
Which one would you like?
Enter [0 .. n-1] or RANDOM>>> 1
X2 L X1 A H L X3 A H
Which one would you like?
Enter [0 .. n-1] or RANDOM>>> 2
X3 L X1 A H L X2 A H
Which one would you like?
Enter [0 .. n-1] or RANDOM>>> UP
```

```
Start Nonterminal>> T
Number of occurrences of (Limit = 12)
X's>> 8
A's>> 4
B's>> 3
There are 58800 structures
Runtime: 2382 msec of CPU

Which one would you like?
Enter [0 .. n-1] or RANDOM>>> RANDOM
X8 B L X7 L X3 B L X1 B L X5 A H H H L X2 A H L X4 A H L X6 A H H
Which one would you like?
Enter [0 .. n-1] or RANDOM>>> UP

Start Nonterminal>> UP

Command> UP

Exit
^C
.
```

APPENDIX D

AN EXAMPLE OF POLYA-REDFIELD ENUMERATION

1. Enumeration under the Cyclic Group. This program reads a number n from the user's terminal, then outputs all sequences of zeros and ones of length n that are distinct under the cyclic group. Each vector output is lexicographically largest in its equivalence class, and the vectors are output in decreasing lexicographic order. For example, when $n = 4$, the outputs are 1111, 1110, 1100, 1010, 1000, 0000. The program also reports the number of distinct elements found, which can be checked against Polya's enumeration formula:

$$Z_n = \frac{1}{n} \sum_{i|n} \phi(i) 2^{n/i}.$$

Here $\phi(i)$ is Euler's phi function; it counts the number of integers less than i that are relatively prime to i.

This program is efficient in that it doesn't scan the entire set of 2^n vectors of zeros and ones. Instead, the program uses an "incorrect" block move instruction to skip over large pieces of the set. One nice consequence of this organization is that the test to reject unwanted vectors is relatively simple (see module "Test and output if good" below).

```
  define max_size = 30
  define quit = 10
program cyclic(tty, output);
  label quit;
  type small_integer = 0 .. max_size;
  var ⟨ Variables used by the program 2 ⟩
    begin while true do  { The program will solicit numerous problem sizes }
      begin write(tty, 'Please␣Enter␣Problem␣Size␣␣'); read(tty, n);
      ⟨ Do it for n 3 ⟩;
    quit: writeln(tty, 'Total␣Number␣of␣Vectors␣Found␣=␣', count : 3,
          '␣␣<ctr>C␣&␣.finish␣to␣see␣file');
      writeln(output, 'N␣=', n : 4, '␣␣␣Count␣=␣', count : 3);
      end;
    end.
```

2. The only data structure is an array *vector* of booleans which the user perceives as an array of 0/1.

```
⟨ Variables used by the program 2 ⟩ ≡
n: small_integer;  { n, the problem size, is read from the terminal }
i: small_integer;  { i is a scratch variable, used for it   ations through vector }
vector: array [small_integer] of boolean;  { the booleans will be printed as zeros or ones }
reset: small_integer;  { the location of the rightmost true in vector }
next_to_copy: small_integer;  { the next boolean in vector to be copied }
count: integer;  { counts number of inequivalent vectors found }
```

This code is used in section 1.

3. The central loop of the program scans the array for the rightmost one, resets this one to zero, and then copies the beginning of the array into the positions following the reset one. This operation is likely to generate an isomorphically distinct vector of zeros and ones.

```
⟨ Do it for n 3 ⟩ ≡
  count ← 0;
  for i ← 0 to n do vector[i] ← true;
  ⟨ Print the array 4 ⟩;
  vector[n] ← false;  { Hereafter the last position in vector is fixed at zero }
  ⟨ Print the array 4 ⟩;
  while true do
    begin ⟨ Find the rightmost one and reset it; goto quit if all zeros 5 ⟩;
    ⟨ Copy from the beginning of the vector to the zeros after reset 6 ⟩;
    ⟨ Test and output if good 7 ⟩;
    end;
```

This code is used in section 1.

4. Convert the boolean array *vector* to 0/1 and print the array in *output*.

```
⟨ Print the array 4 ⟩ ≡
  begin for i ← 1 to n do
    if vector[i] then write('1')
    else write('0');
  count ← count + 1; writeln;
  end;
```

This code is used in sections 3, 3, and 7.

5. The rightmost one in *vector* is set to zero. The variable *reset* points to the changed location. When *vector*[1 .. *n*] is all zero, the otherwise useless one entry at *vector*[0] forces an exit.

```
⟨ Find the rightmost one and reset it; goto quit if all zeros 5 ⟩ ≡
  reset ← n;
  repeat reset ← reset − 1;
  until vector[reset];
  if reset = 0 then goto quit;
  vector[reset] ← false;
```

This code is used in section 3.

6. This is a block move instruction with origin 1 and destination *reset* + 1. It works "incorrectly" in the sense that the origin area may overlap the destination area; since the direction of copying is forward, one entry may be replicated several times. This is precisely the behavior desired for the algorithm.

```
⟨ Copy from the beginning of the vector to the zeros after reset 6 ⟩ ≡
  next_to_copy ← 1; i ← reset + 1;
  while i ≠ n do
    begin vector[i] ← vector[next_to_copy]; i ← i + 1; next_to_copy ← next_to_copy + 1;
    end;
```

This code is used in section 3.

7. The newly created vector is "good" if the next item in line for copying is one. This item would normally land in the last entry of *vector*, which is permanently fixed at zero, so we know that the first part of *vector* is lexicographically larger than the portion of *vector* following *reset*.

In special cases we allow the next item in line for copying to be zero. This corresponds to non-prime n where there is a repeated pattern in the array. The **mod** below checks for this posibility.

```
⟨ Test and output if good 7 ⟩ ≡
  if vector[next_to_copy] ∨ (n mod reset = 0) then
    begin ⟨ Print the array 4 ⟩
    end;
```

This code is used in section 3.

BIBLIOGRAPHY

[Aho 1972] Alfred V. Aho and Jeffery D. Ullman
The Theory of Parsing, Translation, and Compiling; Volume 1: Parsing
Prentice-Hall, 1972

[André 1879] D. André
Développements de séc x et de tang x.
Comptes Rendus
Hebdomadaires des Séances de L'Academie des Sciences 88:965-967, 1879

[Andrews 1971] George E. Andrews
On the Foundations of Combinatorial Theory V,
Eulerian Differential Operators
Studies in Applied Mathematics, 50(4):345-375, 1971

[Burge 1972] William H. Burge
An Analysis of a Tree Sorting Method and
Some Properties of a Set of Trees
First USA-JAPAN Computer Conference, 372-378, 1972

[Carlitz 1959] L. Carlitz and J. Riordan
The Number of Labeled Two-Terminal Series-Parallel Networks
Duke Mathematical Journal 23:435-445, 1959

[Cayley 1889] Arthur Cayley
A Theorem on Trees
Quarterly Journal of Pure and Applied Mathematics 23:376-378, 1889

[Chomsky 1963] N. Chomsky and M. P. Schützenberger
The Algebraic Theory of Context-Free Languages
Computer Programming and Formal Systems,
Studies in Logic and the Foundations of Mathematics, 118–159
North-Holland, 1963

[Comtet 1974] Louis Comtet
Advanced Combinatorics: The Art of Finite and Infinite Expansions
Reidel, 1974

[Cori 1970] Robert Cori
Planar Maps and Bracketing Systems
Combinatorial Structures and Their Applications
Gordon Breach, 1970

[Cori 1972] Robert Cori et Jean Richard
Enumeration des Graphes Planaire a l'Aide
des Series Formelles en Variables Non Commutative
Discrete Mathematics 2:115–162, 1972

[Cori 1975] Robert Cori
Un Code pour les Graphes Planaires et ses Applications
Asterisque, 27, 1975

[Euler 1755] Leonhardo Eulero
Institutiones Calculi Differentialis cum eius vsu
In Analysi Finitorum ac Doctrina Serierum
AcademiæImperialis Scientiarum
Petropolitanæ, 1755

[Flajolet 1980] P. Flajolet, J. Françon, and J. Vuillemin
Sequence of Operation Analysis for Dynamic Data Structures
Journal of Algorithms, 1:111–141, 1980

[Françon 1976] Jean Françon
Arbres Binaires de Recherche: Propriétés Combinatoires et Applications
R.A.I.R.O. Informatique Théorique, 10(12):35–50, 1976

[Goldman 1978] Jay R. Goldman
Formal Languages and Enumeration
Journal of Combinatorial Theory, Series A 24:318–338, 1978

[Goldman 1979] Jay R. Goldman
Formal Languages and Enumeration II
Second International Conference on Combinatorial Mathematics
Annals of the New York Academy of Sciences 319:234–241, 1979

[Gross 1966] Maurice Gross
Application Géométriques des Langages Formels
International Computation Centre Bulletin 5:141–167, 1966

[Harrison 1978] Michael A. Harrison
Introduction to Formal Language Theory
Addison-Wesley, 1978

[Jabotinsky 1947] Eri Jabotinsky
Sur la représentation de la composition de fonctions par un produit de matrices. Application à l'itération de e^x et de $e^x - 1$.
Comptes Rendus
Hebdomadaires des Séances de L'Academie des Sciences 224:323–324, 1947

[Joyal 1981] A. Joyal
Une Théorie Combinatoire des Séries Formelles
Advances in Mathematics 42(1):1–82, 1981

[Knödel 1951] Walter Knödel
Über Zerfällungen
Monatshefte für Mathematik 55(1):20–27, 1951

[Knuth *] Donald E. Knuth
The Art of Computer Programming
Volume 1, Fundamental Algorithms, second edition, 1973
Volume 3, Sorting and Searching, second printing, 1975
Addison-Wesley

[Kuich 1970a] W. Kuich
Enumeration Problems and Context-Free Languages
Combinatorial Theory and its Applications
Colloquia Mathematica Societatis János Bolyai 4:729–735, 1970

[Kuich 1970b] W. Kuich
Languages and the Enumeration of Planted Plane Trees
Indagationes Mathematicæ 32: 268–280, 1970

[Kung 1978] H. T. Kung and J. F. Traub
All Algebraic Functions Can Be Computed Fast
Journal of the Association for Computing Machinery 25(2):245–260, 1978

[MacMahon 1892] P. A. MacMahon
The Combinations of Resistances
The Electrician 28(725):601–602, 1892

[Metropolis 1953] N. Metropolis and S. Ulam
A Property of Randomness of an Arithmetical Function
American Mathematical Monthly, 60:252–253, 1953

[Moon 1970] J. W. Moon
Counting Labelled Trees
Canadian Mathematical Monographs, No. 1
William Clowes and Sons, 1970

[Nijenhuis 1978] Albert Nijenhuis and Herbert S. Wilf
Combinatorial Algorithms for Computers and Calculators
Academic Press, second edition, 1978

[Polya 1937] George Polya
Kombinatorische Anzahlbestimmungen für
Gruppen, Graphen und Chemische Verbindugen
Acta Mathematica, 68:145–254, 1937

[Ramanujan 1919] Srinivasa Ramanujan
Proof of Certain Identities in Combinatory Analysis
Proceedings of the Cambridge Philosophical Society, 19:214–216, 1919

[Raney 1960] George N. Raney
Functional Composition Patterns and Power Series Reversion
Transactions of the American Mathematical Society, 94:441–451, 1960

[Read 1978] Ronald C. Read
Every One a Winner; or, How to Avoid Isomorphism Search
When Cataloguing Combinatorial Configurations
Annals of Discrete Mathematics, 2:107–120, 1978

[Redfield 1927] J. Howard Redfield
The Theory of Group-Reduced Distributions
American Journal of Mathematics, 49:433–455, 1927

[Riordan 1962] John Riordan
Enumeration of Linear Graphs for Mappings of Finite Sets
The Annals of Mathematical Statistics, 33(1):178–185, 1962

[Riordan 1978] John Riordan
An Introduction to Combinatorial Analysis
Princeton University Press, 1978

[Salomaa 1978] Arto Salomaa and Matti Soittola
Automata-Theoretic Aspects of Formal Power Series
Springer-Verlag, 1978

[Schröder 1870] Ernst Schröder
Vier combinatorische Probleme
Zeitschrift für Mathematik und Physik 15:363–376, 1870

[Schützen 1961] M. P. Schützenberger
Some Remarks on Chomsky's Context-Free Languages
Quarterly Progress Report No. 63, October 1961,
Research Laboratory of Electronics,
Massachusetts Institute of Technology

[Sedgewick 1975] Robert Sedgewick
Quicksort
Ph.D. Dissertation, Stanford, 1975

[Stirling 1749] James Stirling
The Differential Method: or, a Treatise concerning Summation and Interpolation of Infinite Series
Translated with the author's approbation by Francis Holliday
Printed for E. Cave at St. John's Gate, London, 1749

[Tutte 1962] W. T. Tutte
A Census of Planar Triangulations
Canadian Journal of Mathematics, 14:21–38, 1962

[Ullman 1980] Jeffrey D. Ullman
Principles of Database Systems
Computer Science Press, 1980

[Viennot 1976] G. Viennot
Quelques Algorithmes de Permutations
Société Mathématique de France, Astérisque 38–39:275–293, 1976

[Wilf 1977] Herbert S. Wilf
A Unified Setting for Sequencing, Ranking and Selection Algorithms for Combinatorial Objects
Advances in Mathematics, 24:281–291, 1977

[Wilf 1978] Herbert S. Wilf
A Unified Setting for Selection Algorithms (II)
Annals of Discrete Mathematics, 2:135–148, 1978

[Williamson 1976] S. G. Williamson
On the Ordering, Ranking, and Random Generation of Basic Combinatorial Sets
Lecture Notes in Computer Science, 579:309–339
Springer-Verlag, 1976

[Zave 1976] Derek A. Zave
A Series Expansion Involving the Harmonic Numbers
Information Processing Letters, 5(3):75–77, 1976